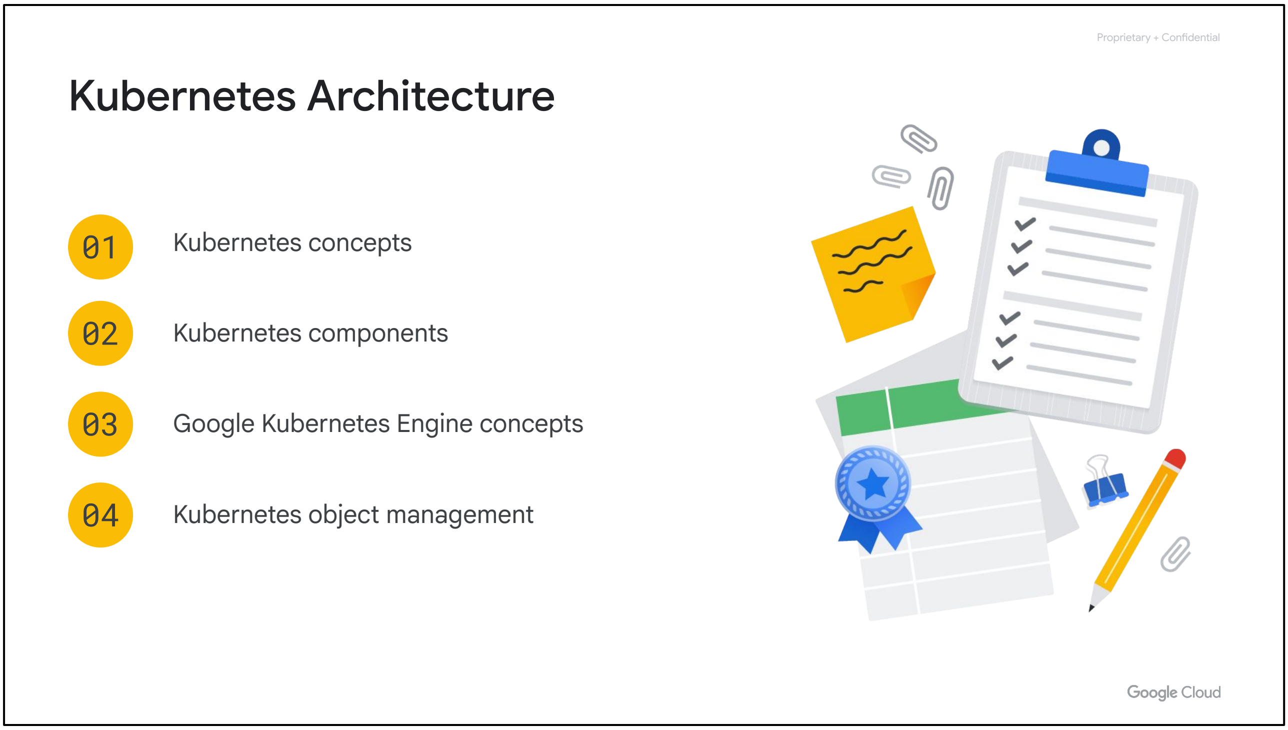
03

Kubernetes Architecture



Proprietary + Confidential

Kubernetes Architecture

01 Kubernetes concepts

02 Kubernetes components

03 Google Kubernetes Engine concepts

04 Kubernetes object management

Google Cloud

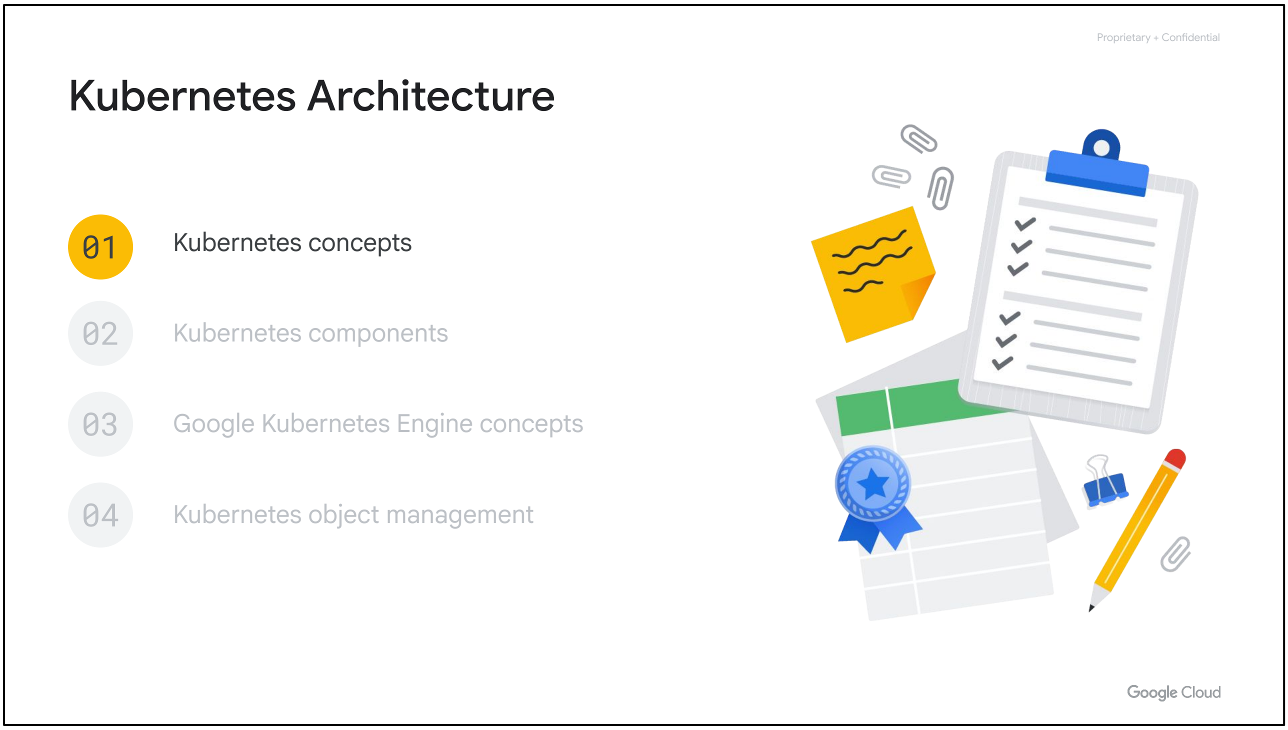
SAY: The third section of the course was designed to answer those questions.

You explore:

- Kubernetes concepts, like Kubernetes object model and the principal of declarative management
- A list of Kubernetes components
- Google Kubernetes Engine concepts, including the Autopilot and standard modes of operation
- And Kubernetes object management

You also get hands-on practice deploying a sample Pod in GKE.

Let’s get started!



Proprietary + Confidential

Kubernetes Architecture

01 Kubernetes concepts

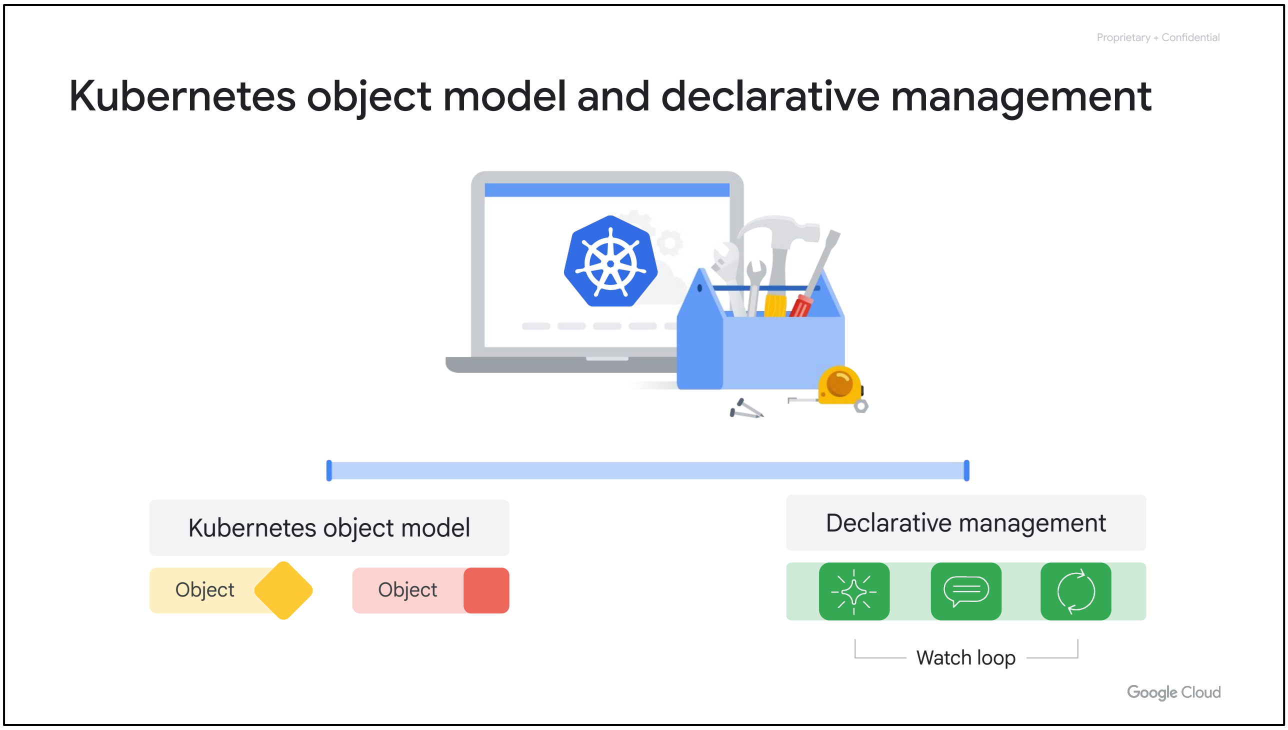
02 Kubernetes components

03 Google Kubernetes Engine concepts

04 Kubernetes object management

Google Cloud

SAY: Let’s begin with some Kubernetes concepts.



Proprietary + Confidential

Kubernetes object model and declarative management

Kubernetes object model

Object

Object

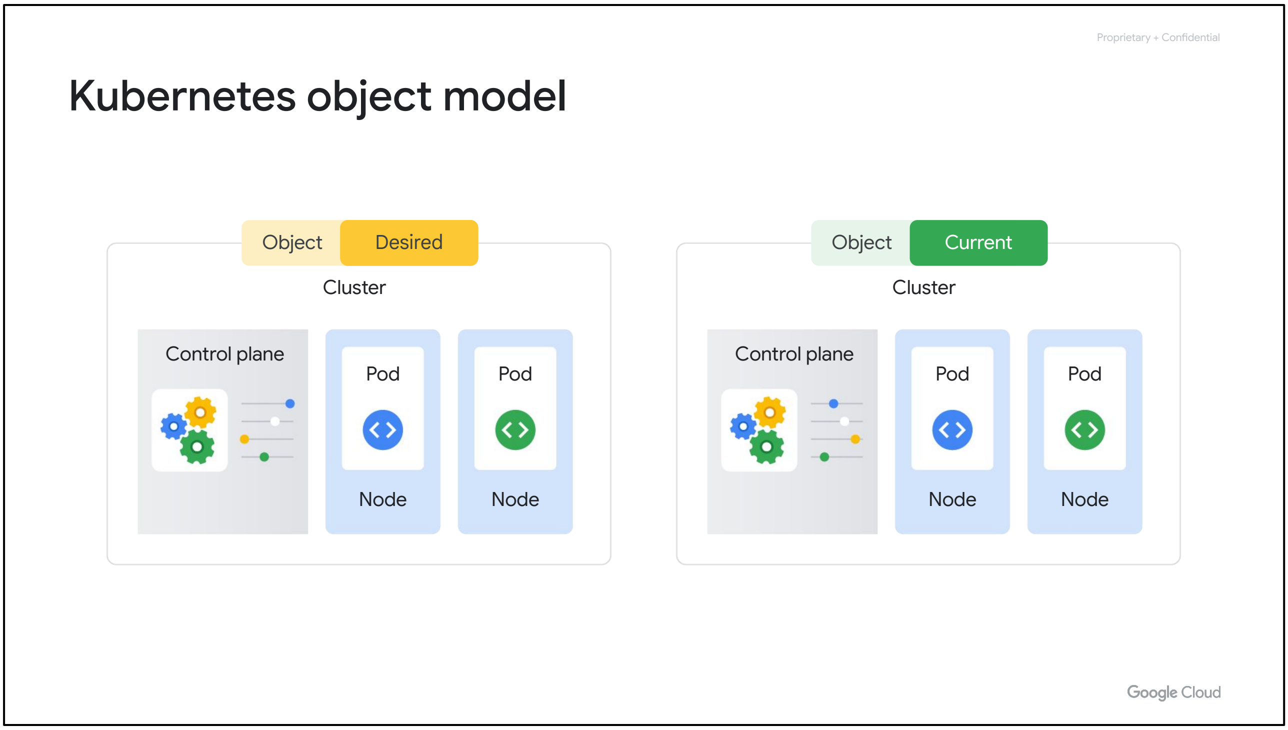
Declarative management

Watch loop

Google Cloud

SAY: To understand how Kubernetes works, it’s important to understand two related concepts.

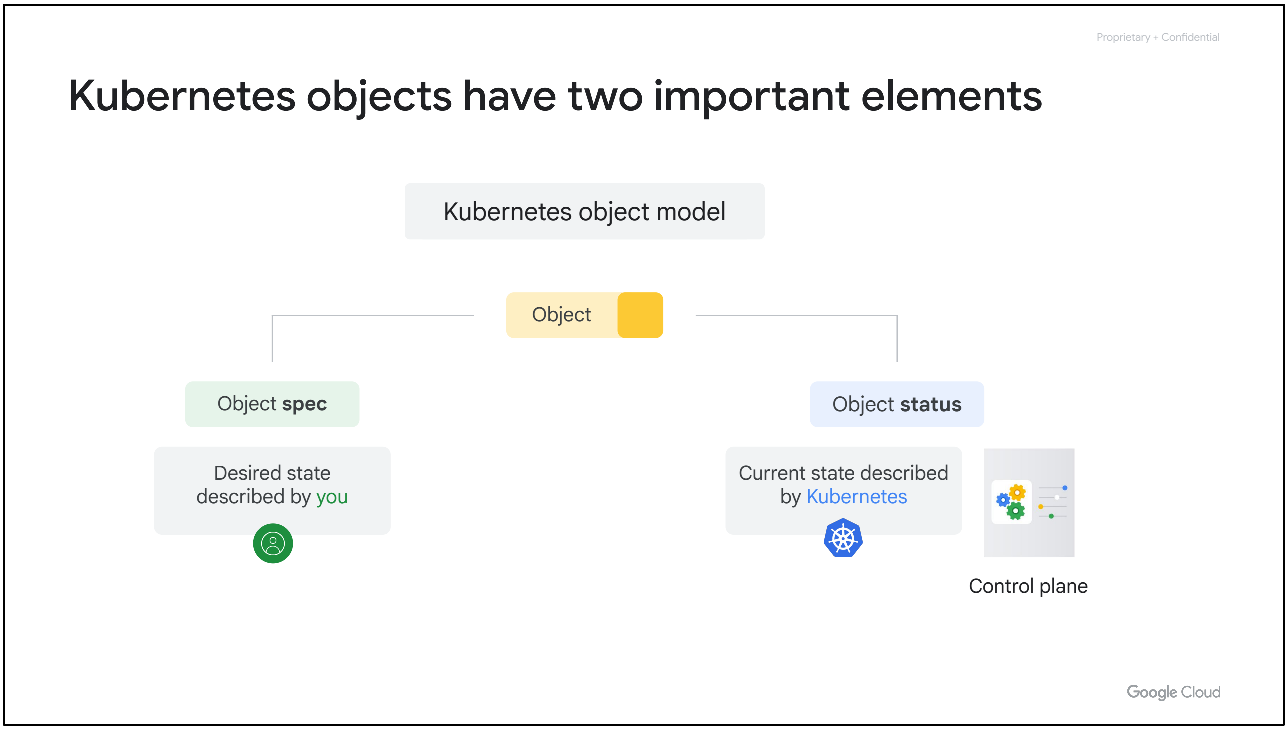
1. The first concept is the **Kubernetes object model**. Each item Kubernetes manages is represented by an object, and you can view and change these objects’ attributes and state.
2. The second concept is the principle of **declarative management**. Kubernetes needs to be told how objects should be managed, and it will work to achieve and maintain that desired state. This is accomplished through a "watch loop."



Kubernetes object model

SAY: A Kubernetes object is defined as a persistent entity that represents the state of something running in a cluster: its *desired state* and its *current state*.

Various kinds of objects represent containerized applications, the resources available to them, and the policies that affect their behavior.



Kubernetes objects have two important elements

Kubernetes object model

Object

Object **spec**

Desired state described by you

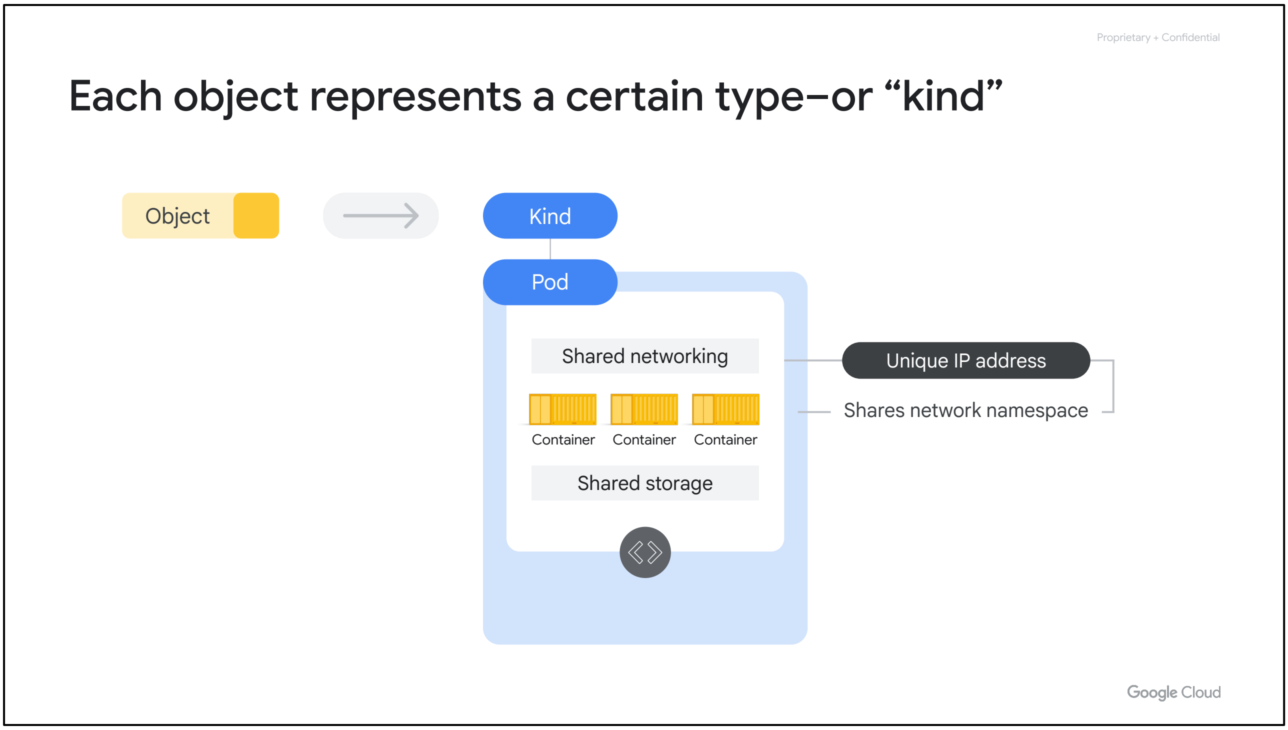
Object **status**

Current state described by Kubernetes

Control plane

SAY: Kubernetes objects have two important elements. The first is an **Object spec** for each object being created. It's here that the desired state of the object is defined by you. The second is the **Object status**, which represents the current state of the object provided by the Kubernetes control plane.

By the way, "Kubernetes control plane" is a term to refer to the various system processes that collaborate to make a Kubernetes cluster work. You'll learn about these processes later.



Each object represents a certain type–or “kind”

Object

Kind

Pod

Shared networking

Container

Container

Container

Shared storage

Unique IP address

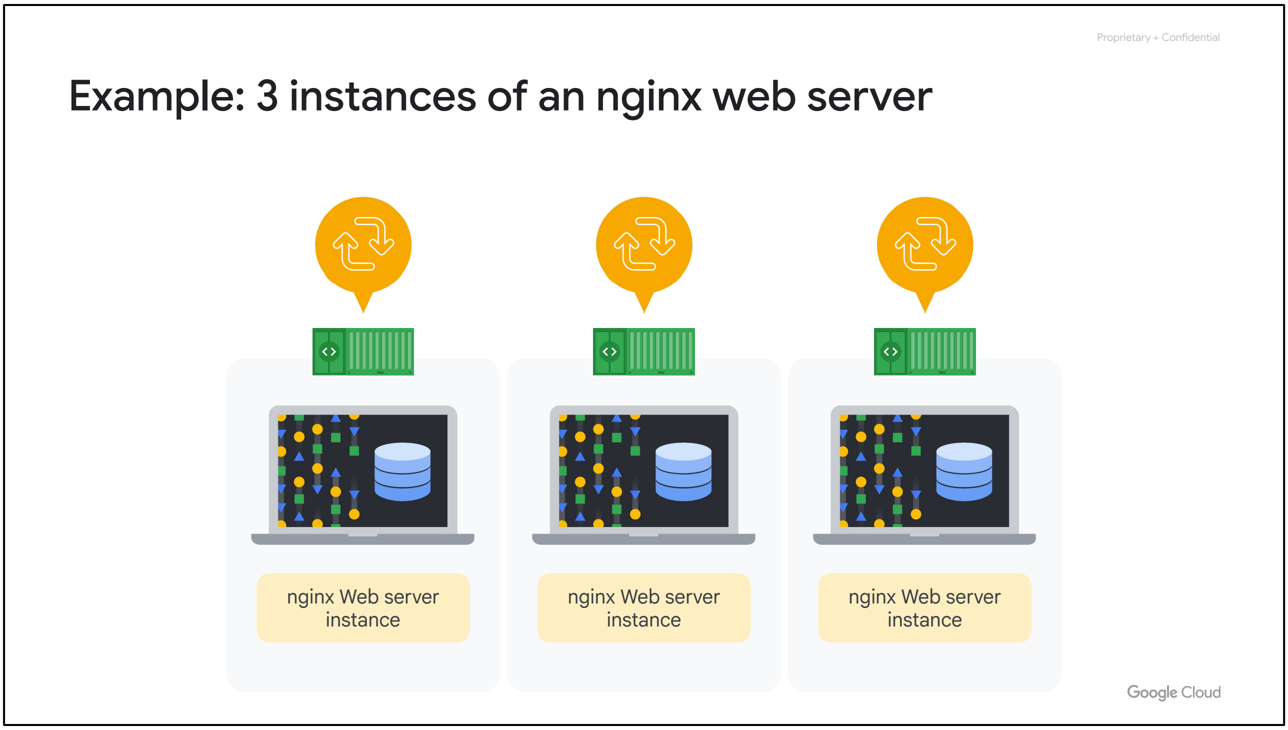
Shares network namespace

SAY: Each object represents a certain type, or “kind,” as it’s referred to in Kubernetes. Pods are the foundational building block of the standard Kubernetes model, and they’re the smallest deployable Kubernetes object. Every running container in a Kubernetes system is in a Pod.

A Pod creates the environment where the containers live, and that environment can accommodate one or more containers. If there is more than one container in a Pod, they are tightly coupled and share resources, like networking and storage.

Kubernetes assigns each Pod a unique IP address, and every container within a Pod shares the network namespace, including IP address and network ports. Containers within the same Pod can communicate through localhost, 127.0.0.1.

A Pod can also specify a set of storage volumes that is shared among its containers.



Proprietary + Confidential

Example: 3 instances of an nginx web server

nginx Web server instance

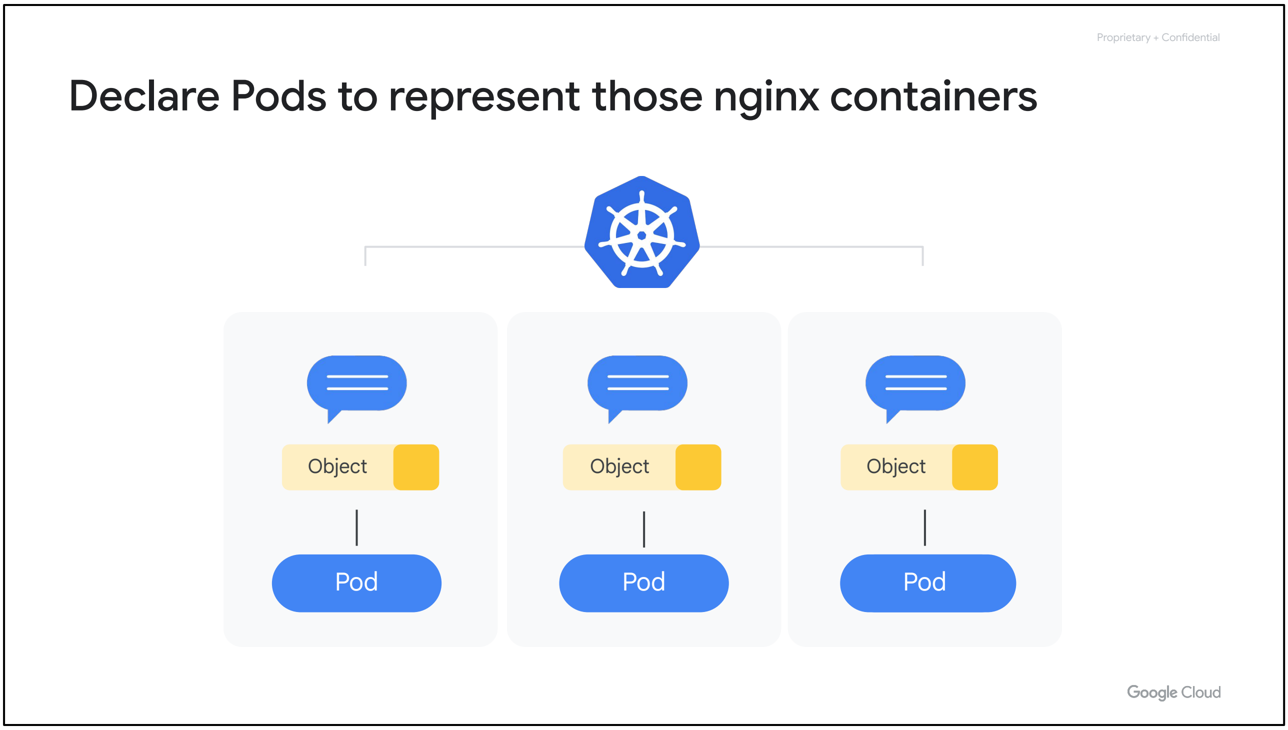
nginx Web server instance

nginx Web server instance

Google Cloud

SAY: Let’s explore an example where you want three instances of an nginx Web server, each in its own container, to be always kept running.

How can this be achieved in Kubernetes?



Declare Pods to represent those nginx containers

Object

Pod

Object

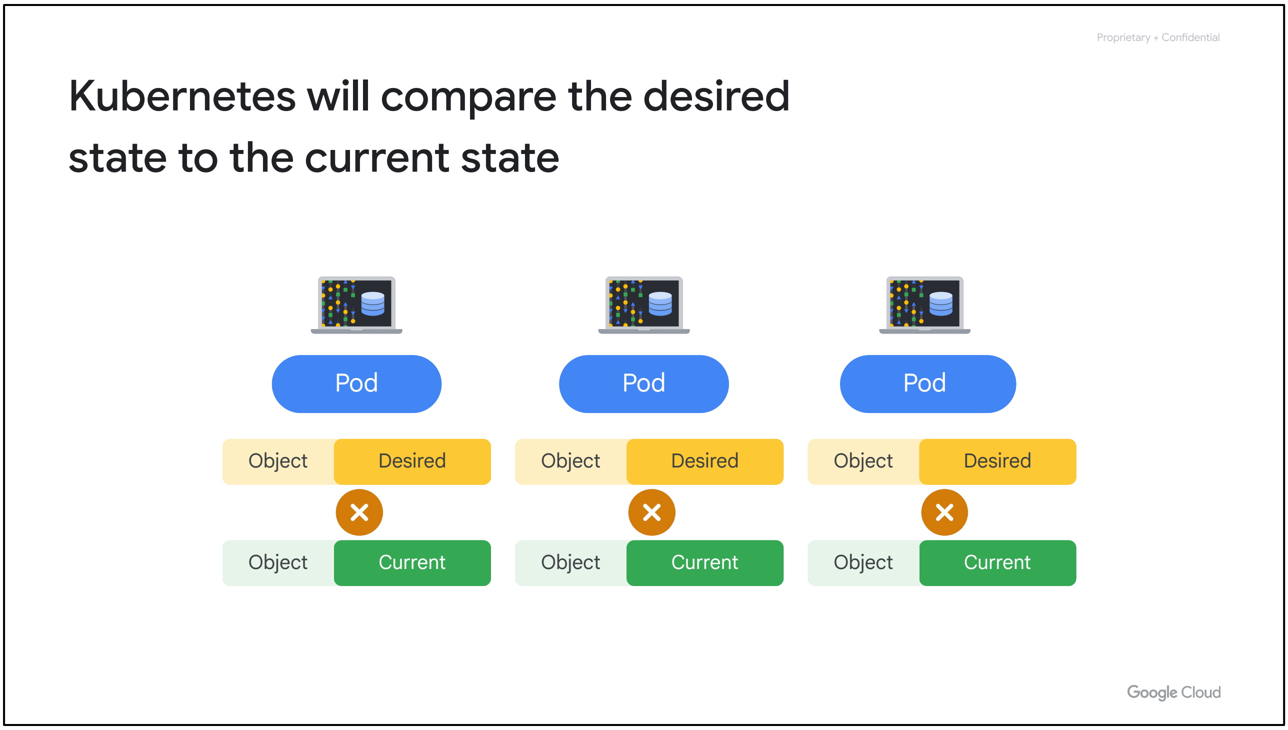
Pod

Object

Pod

SAY: How can this be achieved in Kubernetes? You might recall that Kubernetes operates off of the principle of declarative management, which means that you need to declare some objects to represent those nginx containers, and in this case, those objects should be Pods.

From there, it's Kubernetes's job to launch those Pods and keep them in existence.



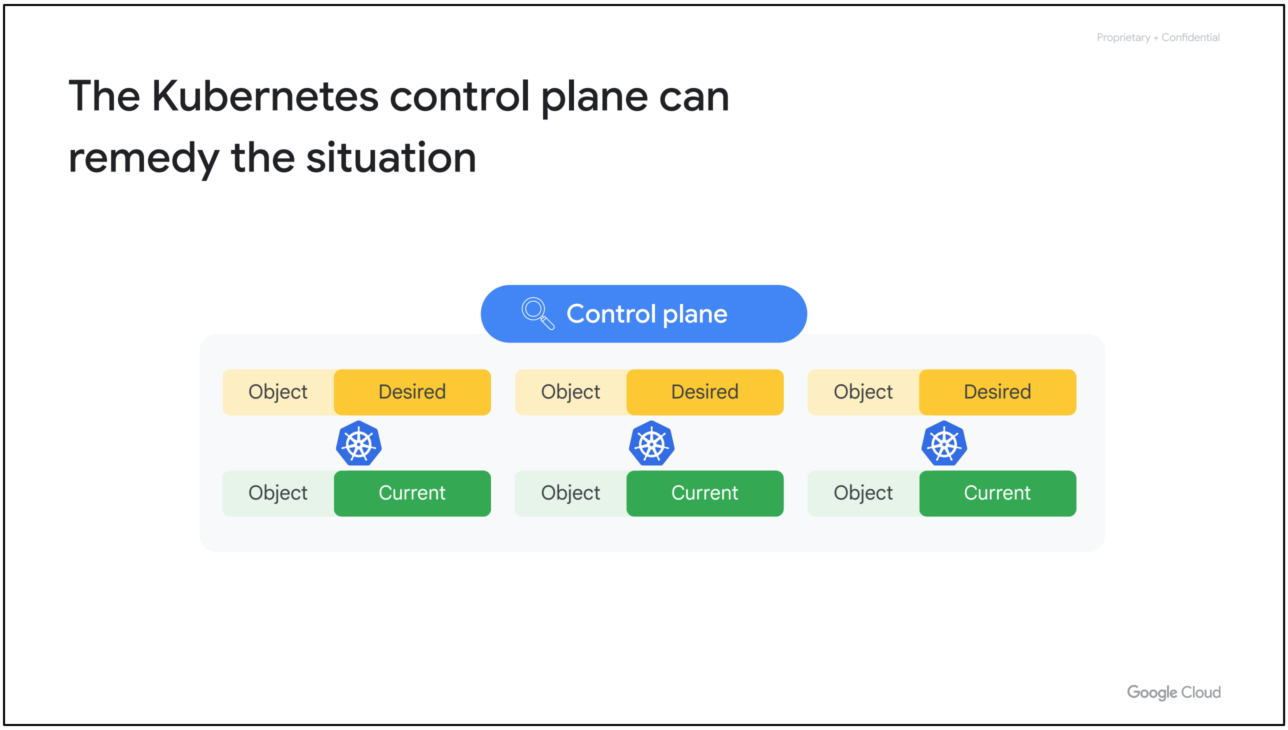
Proprietary + Confidential

Kubernetes will compare the desired state to the current state

Google Cloud

Alright, so you’ve given Kubernetes a desired state that consists of three nginx Pods, to be always kept running. We did this by telling Kubernetes to create and maintain one or more objects that represent them.

Now Kubernetes will compare the *desired state* to the *current state*. For this example, let’s say our declaration of three nginx containers is completely new, meaning that the current state *does not* match the desired state.



The Kubernetes control plane can remedy the situation

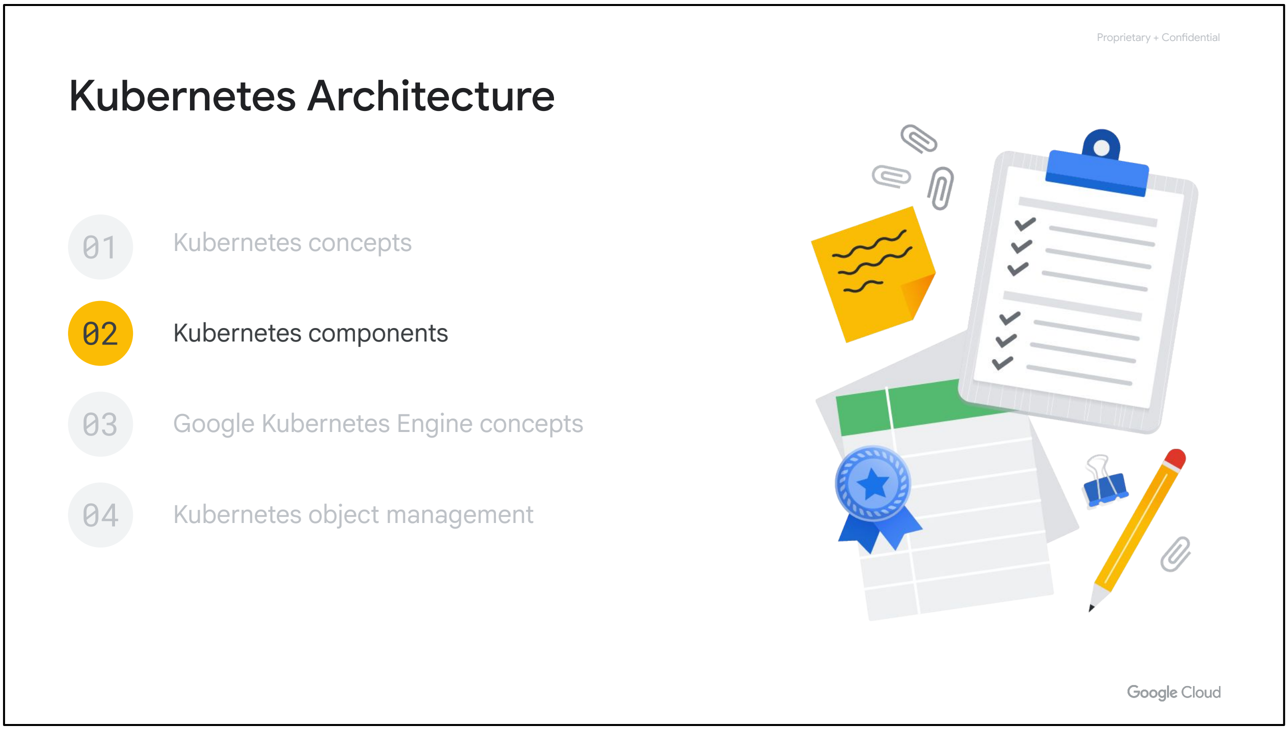
Control plane

Object	Desired	Object	Desired	Object	Desired
Object	Current	Object	Current	Object	Current

Google Cloud

SAY: So Kubernetes, specifically its control plane, can remedy the situation. Because the number of desired Pods that run for the object you declared is 3, and 0 are presently running, 3 will be launched.

And the Kubernetes control plane will continuously monitor the state of the cluster, endlessly comparing reality to what was declared, and remedying the state as needed.



Kubernetes Architecture

01 Kubernetes concepts

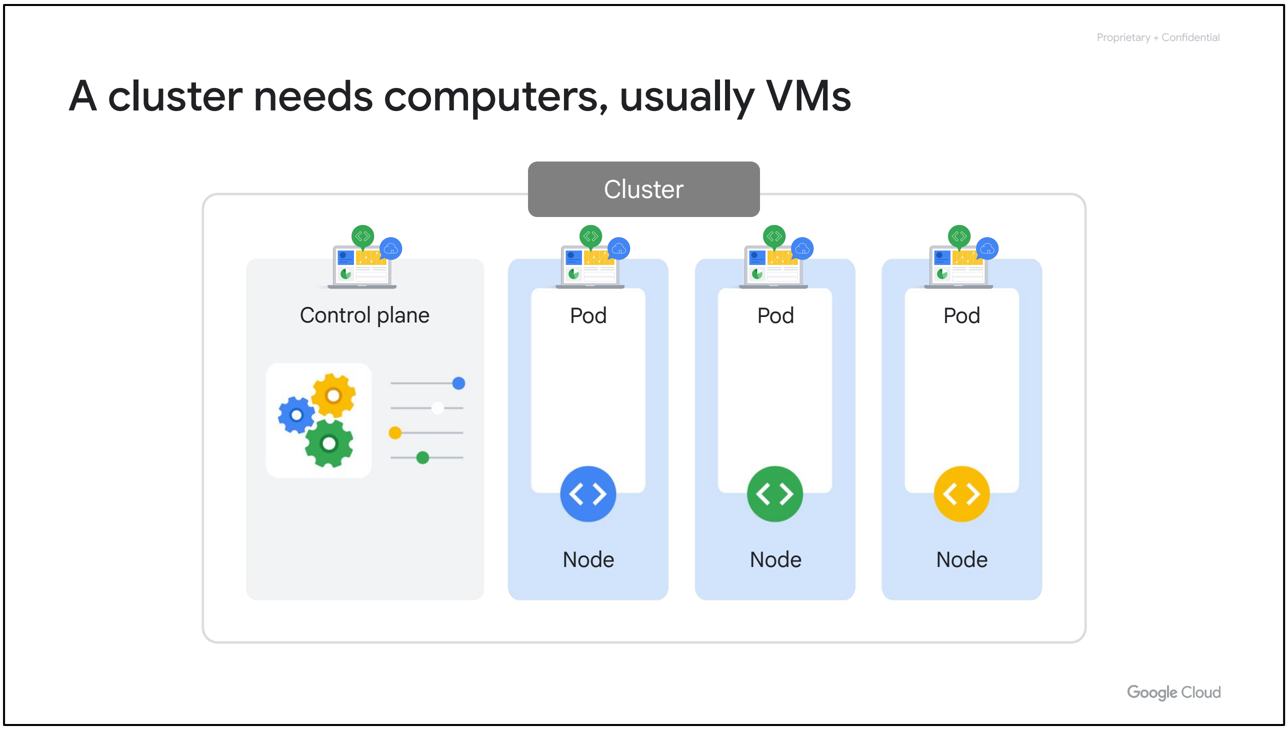
02 Kubernetes components

03 Google Kubernetes Engine concepts

04 Kubernetes object management

SAY: The Kubernetes control plane is the fleet of cooperating processes that make a Kubernetes cluster work. Although you might only directly work with a few of these components, it’s important to understand what the fleet does and the role they each play.

In this section of the course, you’ll see how a Kubernetes cluster is constructed, part by part. This will help you illustrate how a Kubernetes cluster that runs in GKE is easier to manage than one you provisioned yourself.



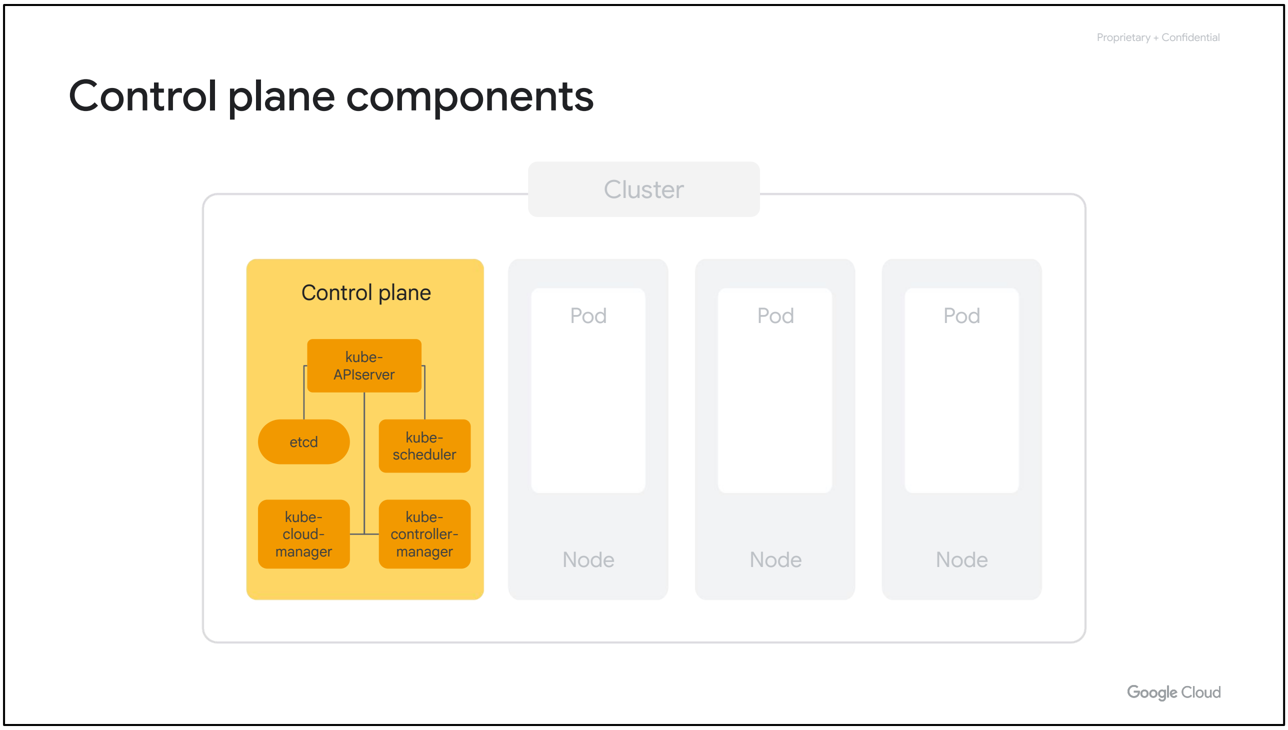
Proprietary + Confidential

A cluster needs computers, usually VMs

Google Cloud

SAY: First, a cluster needs computers, and these computers are usually virtual machines. They always are in GKE, but they could be physical computers too.

One computer is called the *control plane*, and the others are called *nodes*. The node's job is to run Pods, and the control plane's is to coordinate the entire cluster.



Control plane components

Cluster

Control plane

kube-APIserver

etcd

kube-scheduler

kube-cloud-manager

kube-controller-manager

Pod

Node

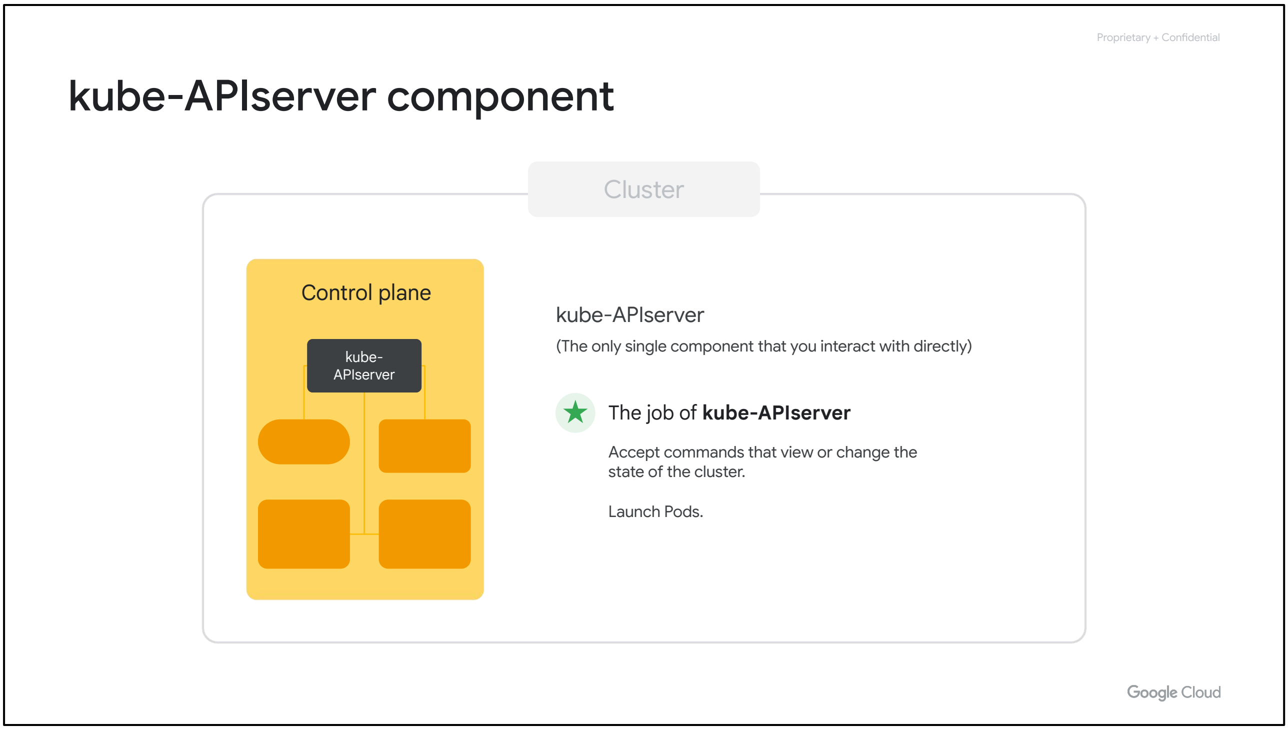
Pod

Node

Pod

Node

SAY: Let’s look at the control-plane components. Several critical Kubernetes components run on the control plane.



kube-APIserver component

kube-APIserver

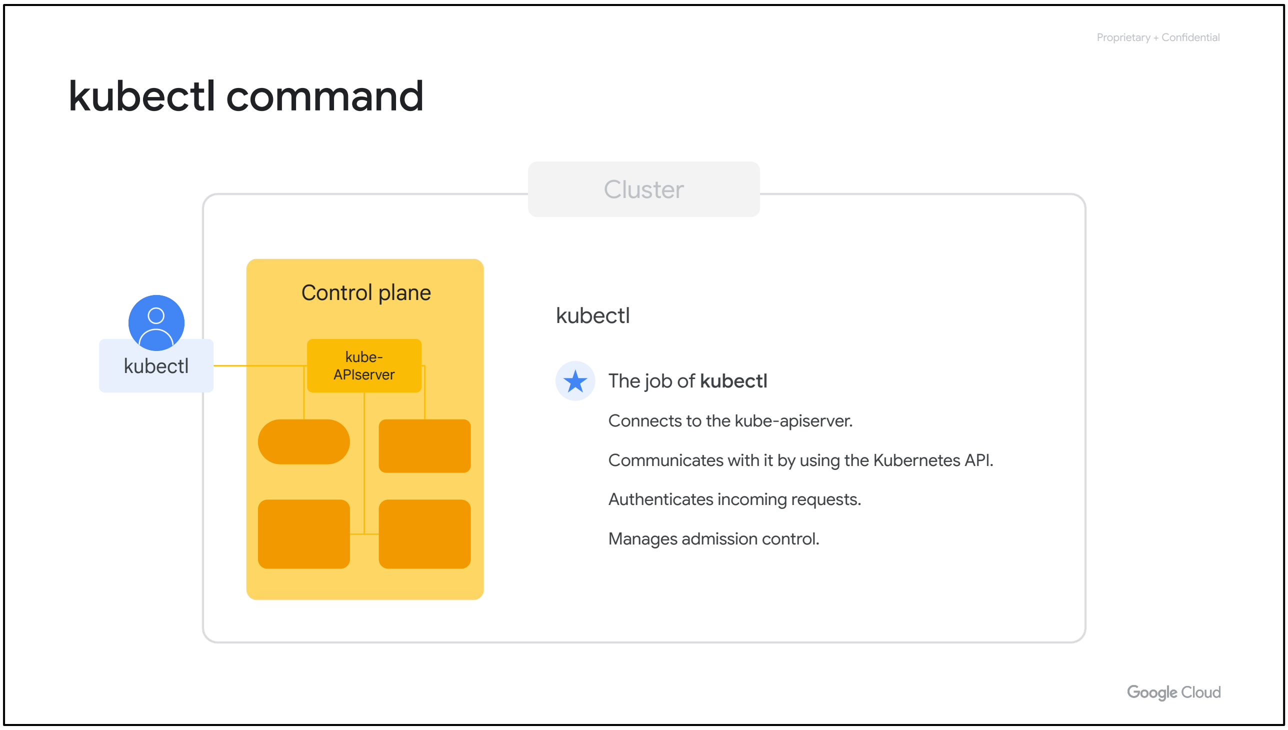
(The only single component that you interact with directly)

★ The job of **kube-APIserver**

Accept commands that view or change the state of the cluster.

Launch Pods.

SAY: First is the **kube-apiserver** component, which is the only single component that you interact with directly. The job of this component is to accept commands that view or change the state of the cluster. This includes launching Pods.



kubectl command

kubectl

★ The job of **kubectl**

Connects to the kube-apiserver.

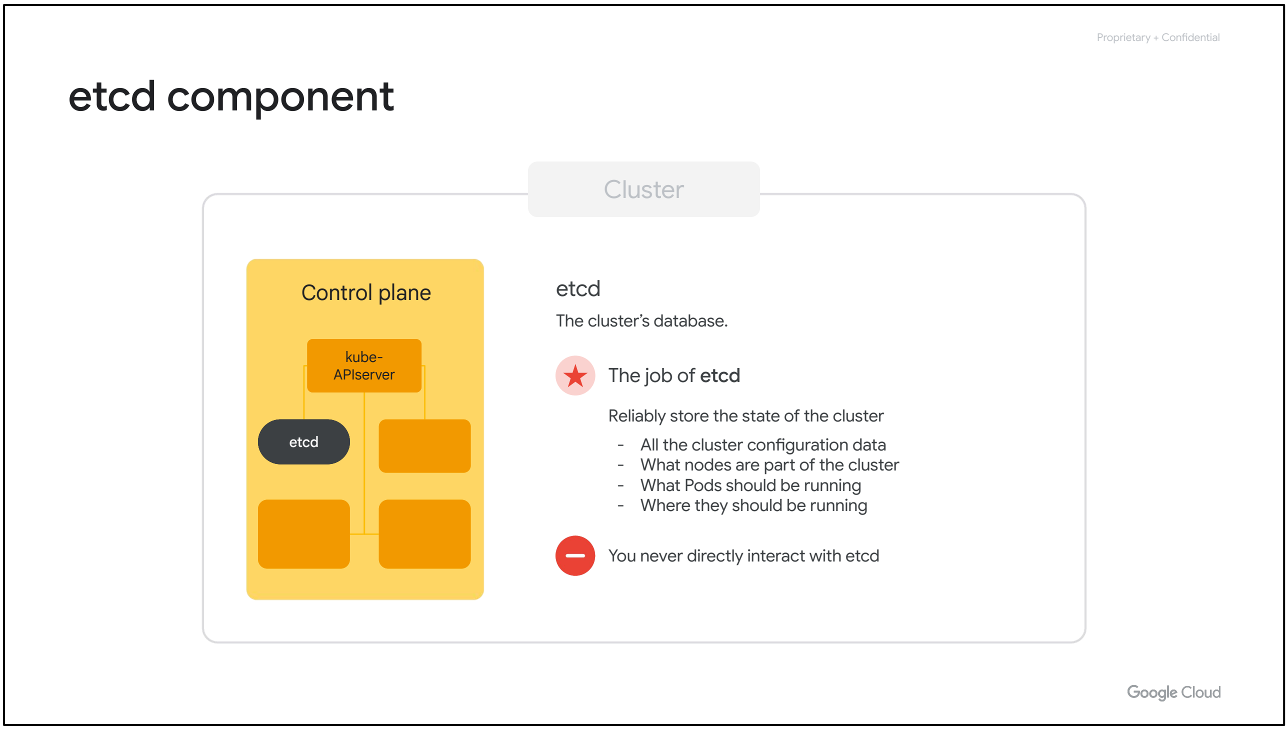
Communicates with it by using the Kubernetes API.

Authenticates incoming requests.

Manages admission control.

Google Cloud

SAY: Next is the **kubectl** command. The job of the *kubectl* command is to connect to the kube-apiserver and communicate with it using the Kubernetes API. The kube-apiserver also authenticates incoming requests, determines whether they are authorized and valid, and manages admission control. But it's not just kubectl that talks with kube-apiserver. In fact, any query or change to the cluster's state must be addressed to the kube-apiserver.



etcd component

Cluster

Control plane

kube-APIserver

etcd

etcd

The cluster's database.

The job of **etcd**

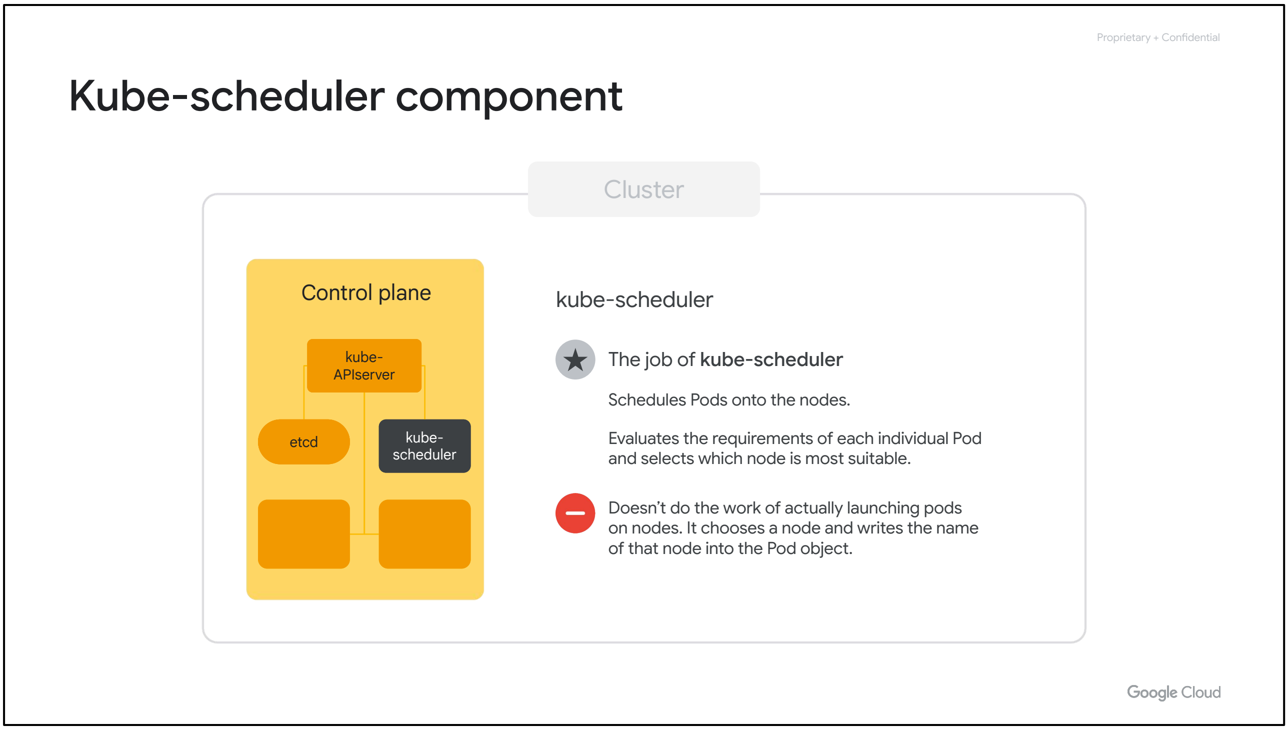
Reliably store the state of the cluster

- All the cluster configuration data
- What nodes are part of the cluster
- What Pods should be running
- Where they should be running

You never directly interact with etcd

SAY: There is the **etcd** component, which is the cluster's database. Its job is to reliably store the state of the cluster. This includes all the cluster configuration data, along with more dynamic information such as what nodes are part of the cluster, *what* Pods should be running, and *where* they should be running.

You'll never directly interact with etcd, instead the kube-apiserver interacts with the database on behalf of the rest of the system.



Kube-scheduler component

kube-scheduler

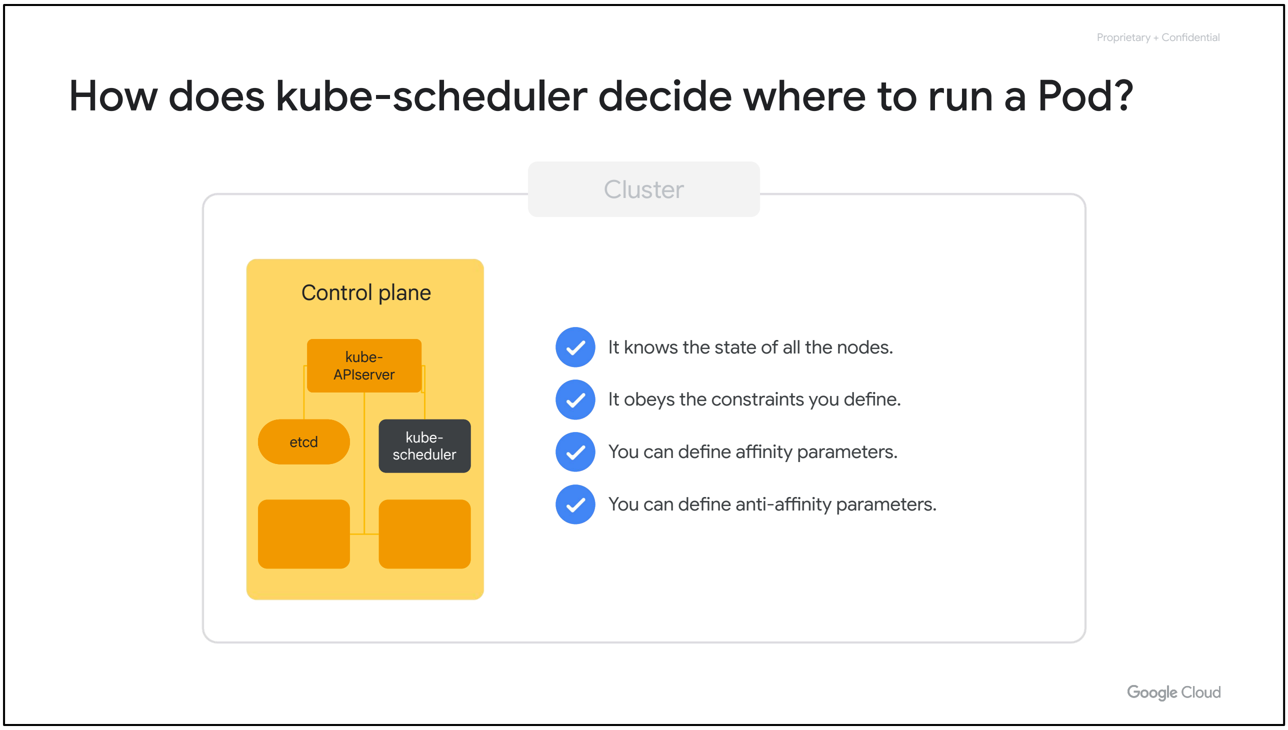
★ The job of **kube-scheduler**

Schedules Pods onto the nodes.

Evaluates the requirements of each individual Pod and selects which node is most suitable.

⊖ Doesn't do the work of actually launching pods on nodes. It chooses a node and writes the name of that node into the Pod object.

SAY: Next is **kube-scheduler**, which is responsible for scheduling Pods onto the nodes. Kube-scheduler evaluates the requirements of each individual pod and selects which node is most suitable. However, it doesn't do the work of actually launching Pods on nodes (that's done by another component). Instead, whenever it discovers a pod object that doesn't yet have an assigned node, it chooses a node and writes the name of that node into the Pod object.



How does kube-scheduler decide where to run a Pod?

Cluster

Control plane

kube-APIserver

etcd

kube-scheduler

✓ It knows the state of all the nodes.

✓ It obeys the constraints you define.

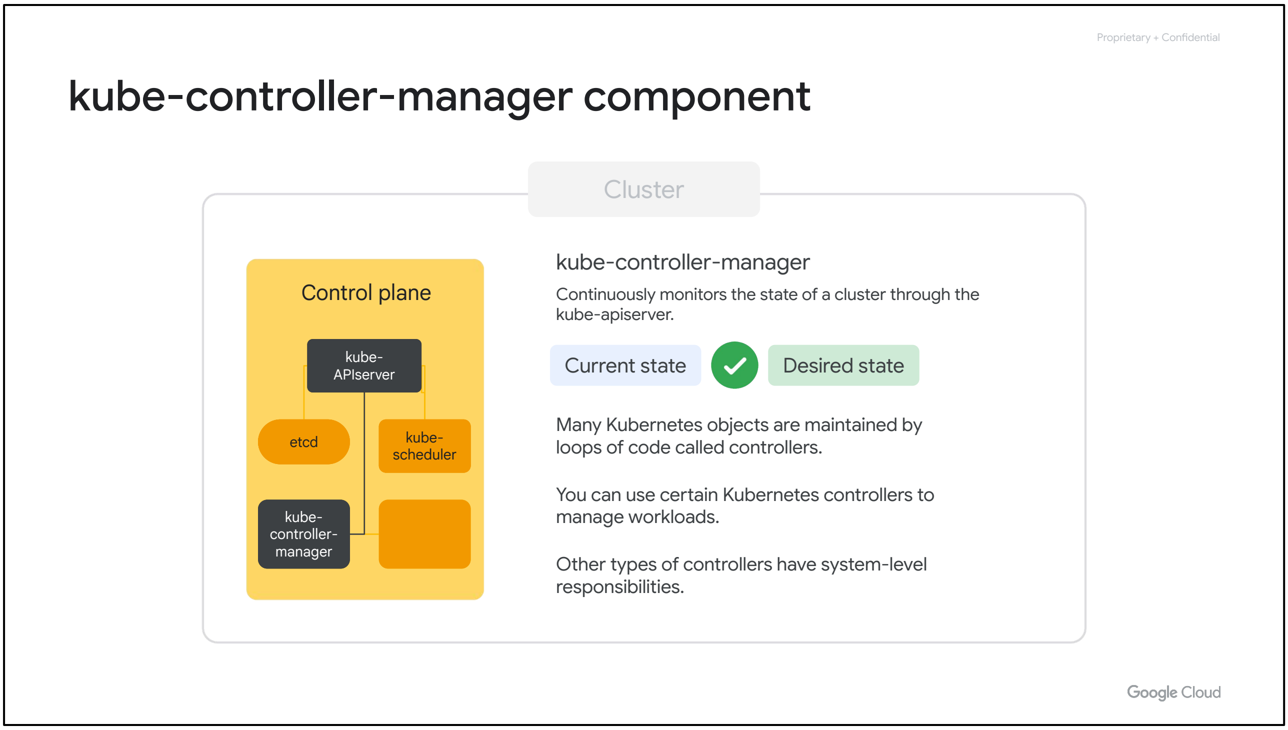
✓ You can define affinity parameters.

✓ You can define anti-affinity parameters.

SAY: How does kube-scheduler decide where to run a Pod?

It knows the state of all the nodes, and also obeys constraints you define regarding where a Pod can run, considering hardware, software, and policy details. For example, you might specify that a certain pod is only allowed to run on nodes with a specific amount of memory.

You can also define affinity parameters, which specify when groups of Pods should run on the same node. Alternatively, you can define anti-affinity parameters, which ensure that Pods do not run on the same node.



kube-controller-manager component

Cluster

Control plane

kube-APIserver

etcd

kube-scheduler

kube-controller-manager

kube-controller-manager

Continuously monitors the state of a cluster through the kube-apiserver.

Current state ✓ Desired state

Many Kubernetes objects are maintained by loops of code called controllers.

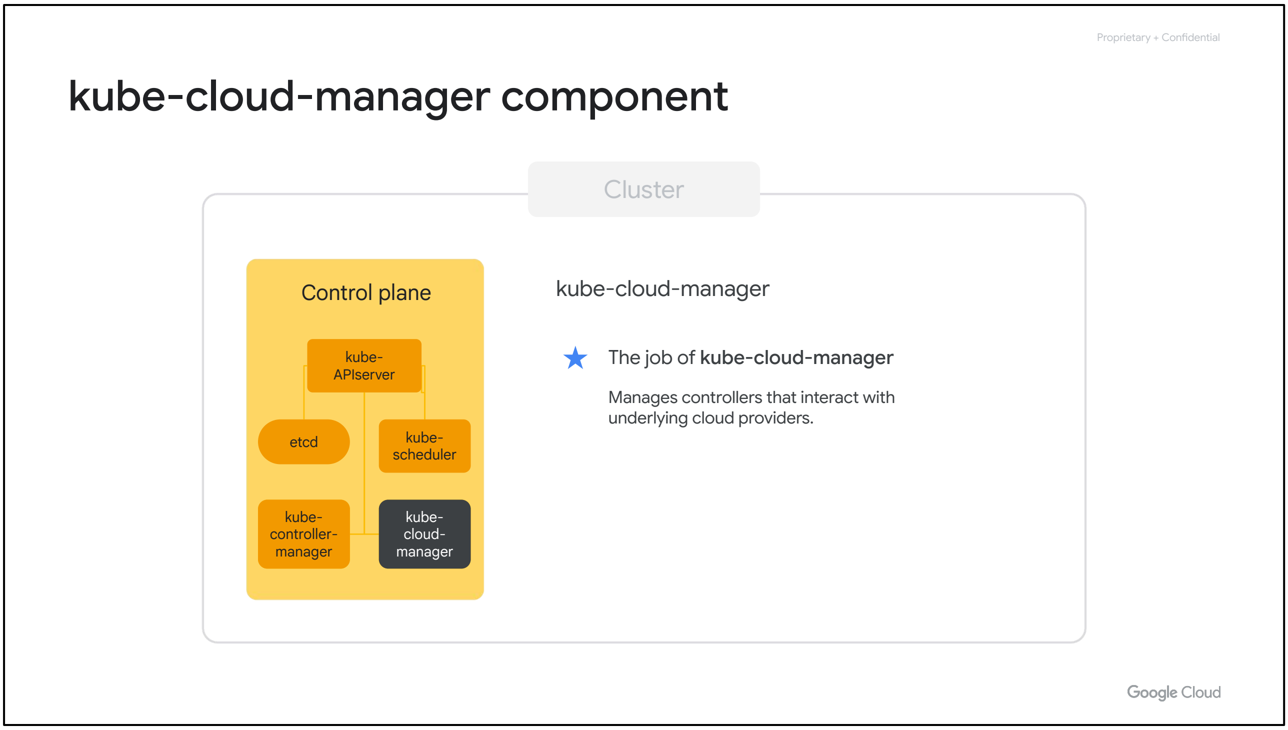
You can use certain Kubernetes controllers to manage workloads.

Other types of controllers have system-level responsibilities.

SAY: The **kube-controller-manager** component has a broader job: it continuously monitors the state of a cluster through the **kube-apiserver**.

Whenever the current state of the cluster doesn't match the desired state, kube-controller-manager will attempt to make changes to achieve the desired state.

- It's called the controller manager because many Kubernetes objects are maintained by loops of code called controllers, which handle the process of remediation.
- You can use certain Kubernetes controllers to manage workloads. For example, remember our problem of keeping 3 nginx Pods always running? They can be gathered into a controller object called a deployment that runs, scales, and brings them together underneath a front end.
- Other types of controllers have system-level responsibilities. For example, the node controller's job is to monitor and respond when a node is offline.



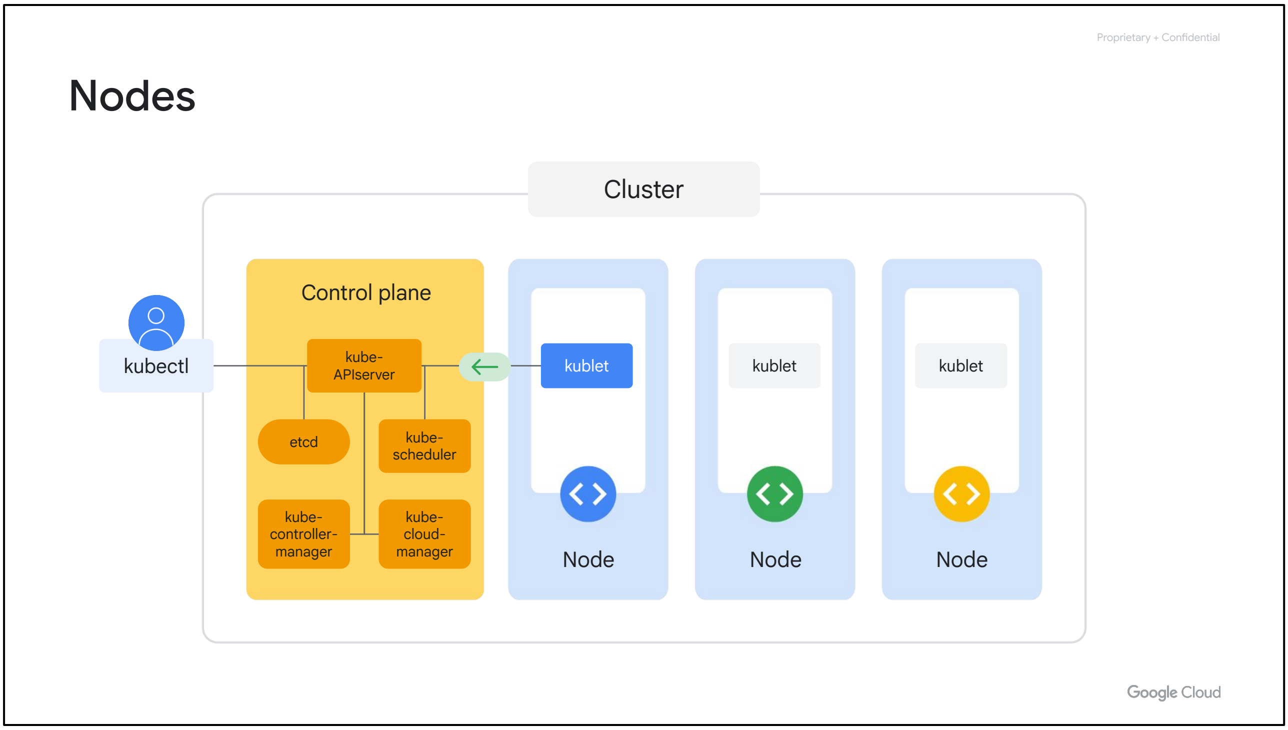
kube-cloud-manager component

kube-cloud-manager

★ The job of **kube-cloud-manager**

Manages controllers that interact with underlying cloud providers.

SAY: The **kube-cloud-manager** component manages controllers that interact with underlying cloud providers. For example, if you manually launched a Kubernetes cluster on Compute Engine, kube-cloud-manager is responsible for bringing in Google Cloud features like load balancers and storage volumes.



Proprietary + Confidential

Nodes

Cluster

kubectl

Control plane

kube-APIserver

etcd

kube-scheduler

kube-controller-manager

kube-cloud-manager

kublet

Node

kublet

Node

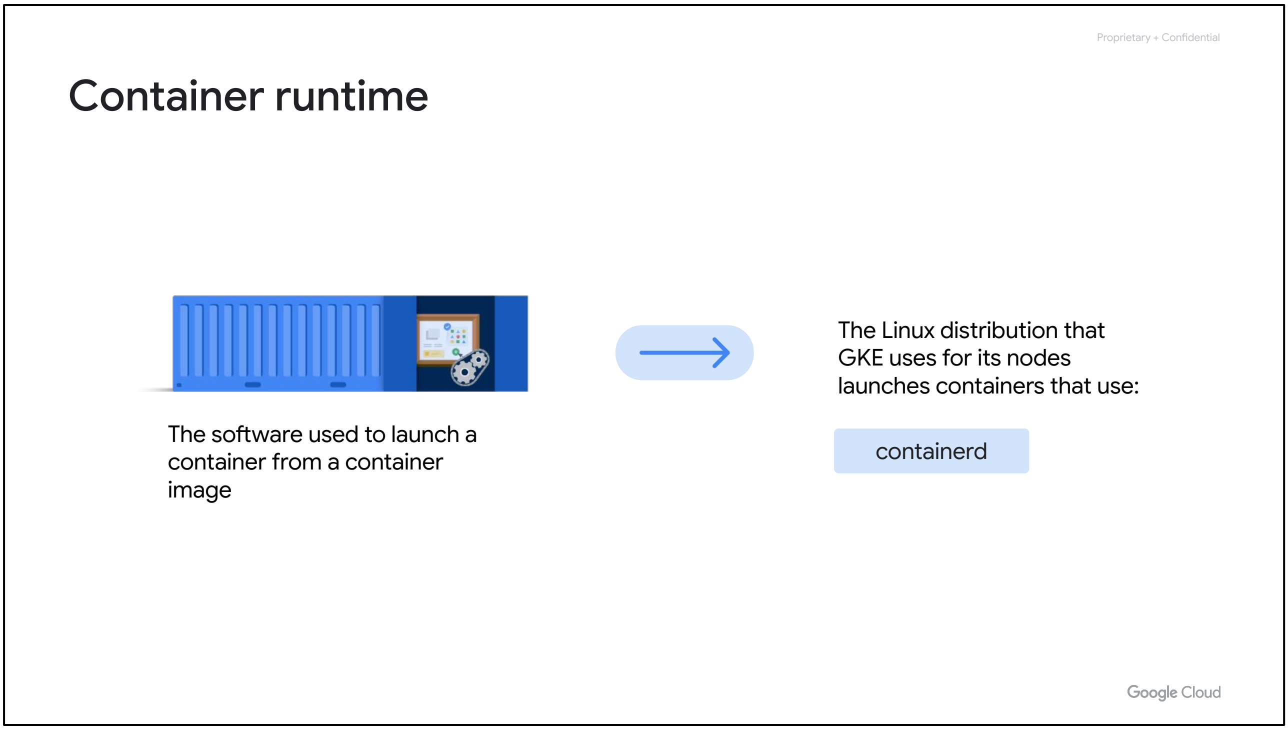
kublet

Node

Google Cloud

SAY: Now let’s shift our focus to nodes.

Each node runs a small family of control-plane components called a kubelet. You can think of a kubelet as Kubernetes’s agent on each node. When the kube-apiserver wants to start a pod on a node, it connects to that node’s kubelet. Kubelet uses the container runtime to start the pod and monitors its lifecycle, including readiness and liveness probes, and reports back to the kube-apiserver.



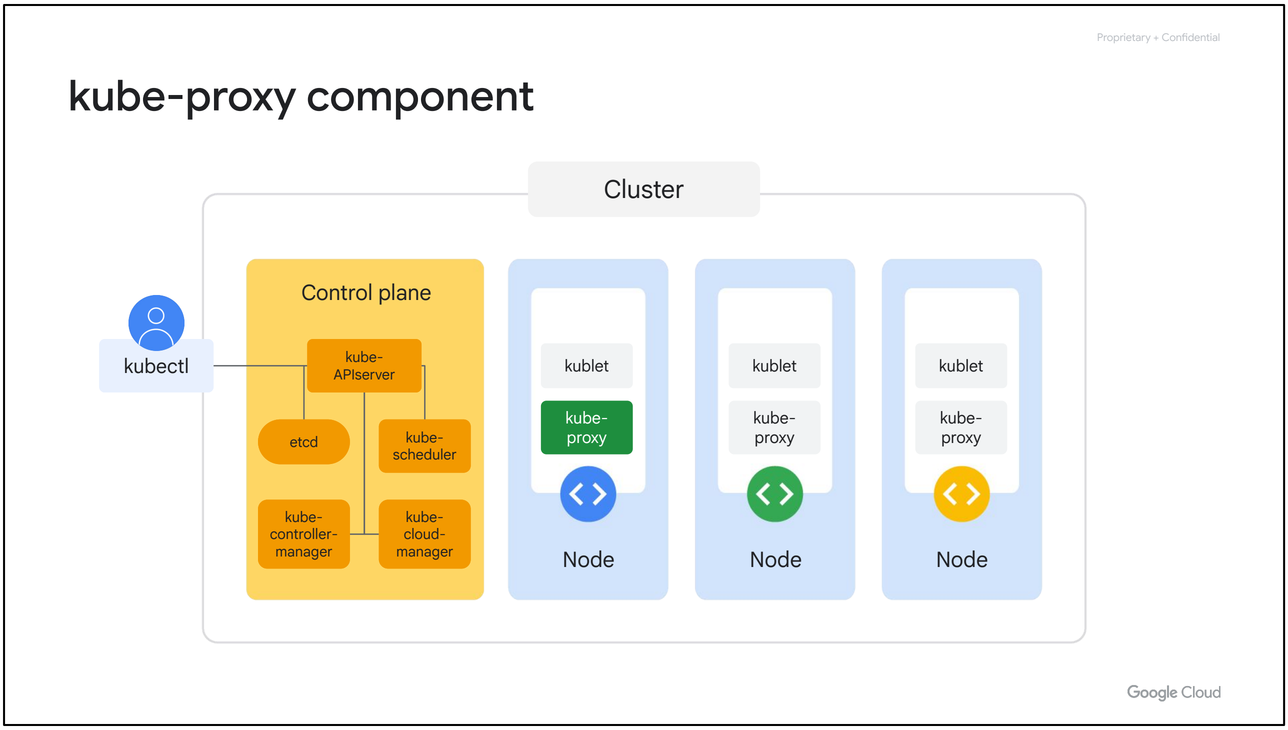
Container runtime

The software used to launch a container from a container image

The Linux distribution that GKE uses for its nodes launches containers that use:

containerd

SAY: The term container runtime, which was mentioned earlier in this course, is the software used to launch a container from a container image. Kubernetes offers several container runtime choices, but the Linux distribution that GKE uses for its nodes launches containers that use **containerd**, the runtime component of Docker.

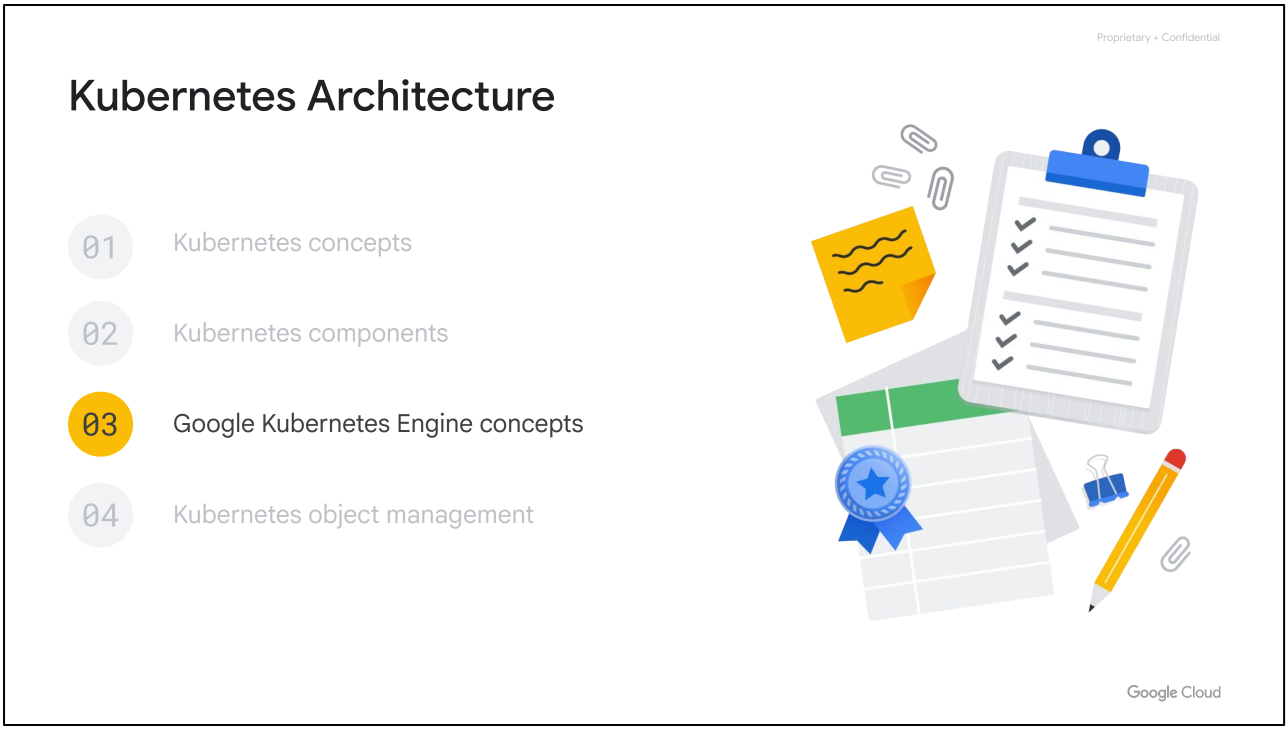


kube-proxy component

Google Cloud

SAY: And finally, there is the **kube-proxy** component, which maintains network connectivity among the Pods in a cluster.

In open source Kubernetes, network connectivity is accomplished by using the firewalling capabilities of iptables, which are built into the Linux kernel.



Proprietary + Confidential

Kubernetes Architecture

01 Kubernetes concepts

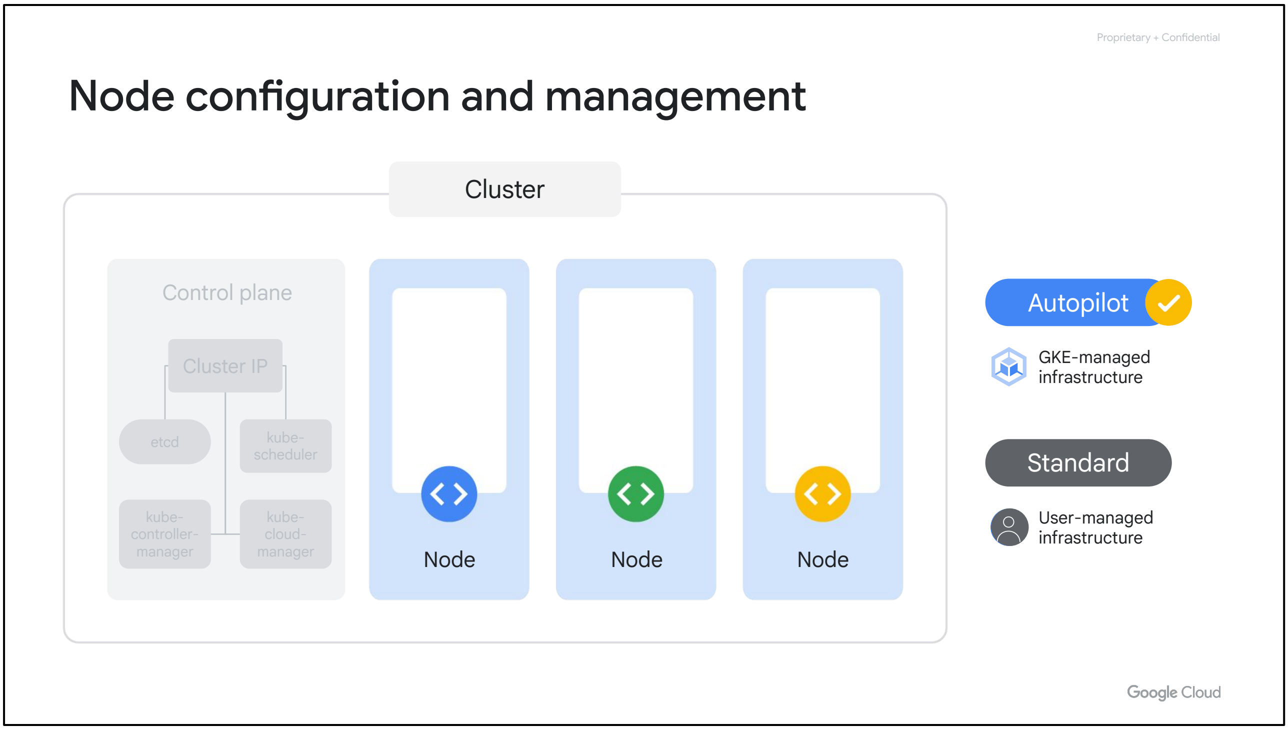
02 Kubernetes components

03 Google Kubernetes Engine concepts

04 Kubernetes object management

Google Cloud

SAY: Let’s focus a bit more on GKE concepts, specifically with a deeper exploration of Autopilot and Standard modes.



Proprietary + Confidential

Node configuration and management

Cluster

Control plane

Cluster IP

etcd

kube-scheduler

kube-controller-manager

kube-cloud-manager

Node

Node

Node

Autopilot ✓

GKE-managed infrastructure

Standard

User-managed infrastructure

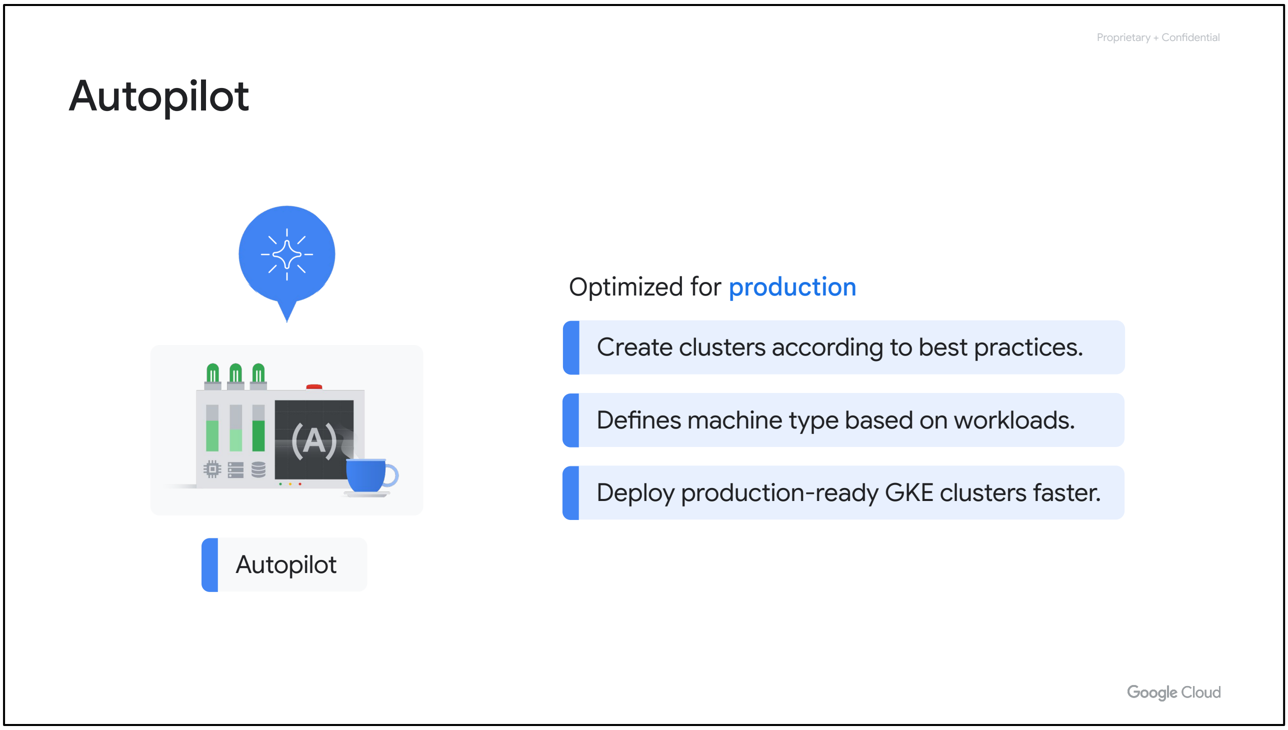
Google Cloud

SAY: We saw that the Kubernetes Control Plane is a complex management system. But how is GKE different from Kubernetes? From the user's perspective, it's a lot simpler. GKE manages all the control plane components for us. It still exposes an IP address to which we send all of our Kubernetes API requests, but GKE is responsible for provisioning and managing all the control plane infrastructure behind it. It also eliminates the need for a separate control plane.

Node configuration and management depends on the type of GKE mode you use.

With the Autopilot mode, which is recommended, GKE manages the underlying infrastructure such as node configuration, autoscaling, auto-upgrades, baseline security configurations, and baseline networking configuration.

With the Standard mode, you manage the underlying infrastructure, including configuring the individual nodes.



Autopilot

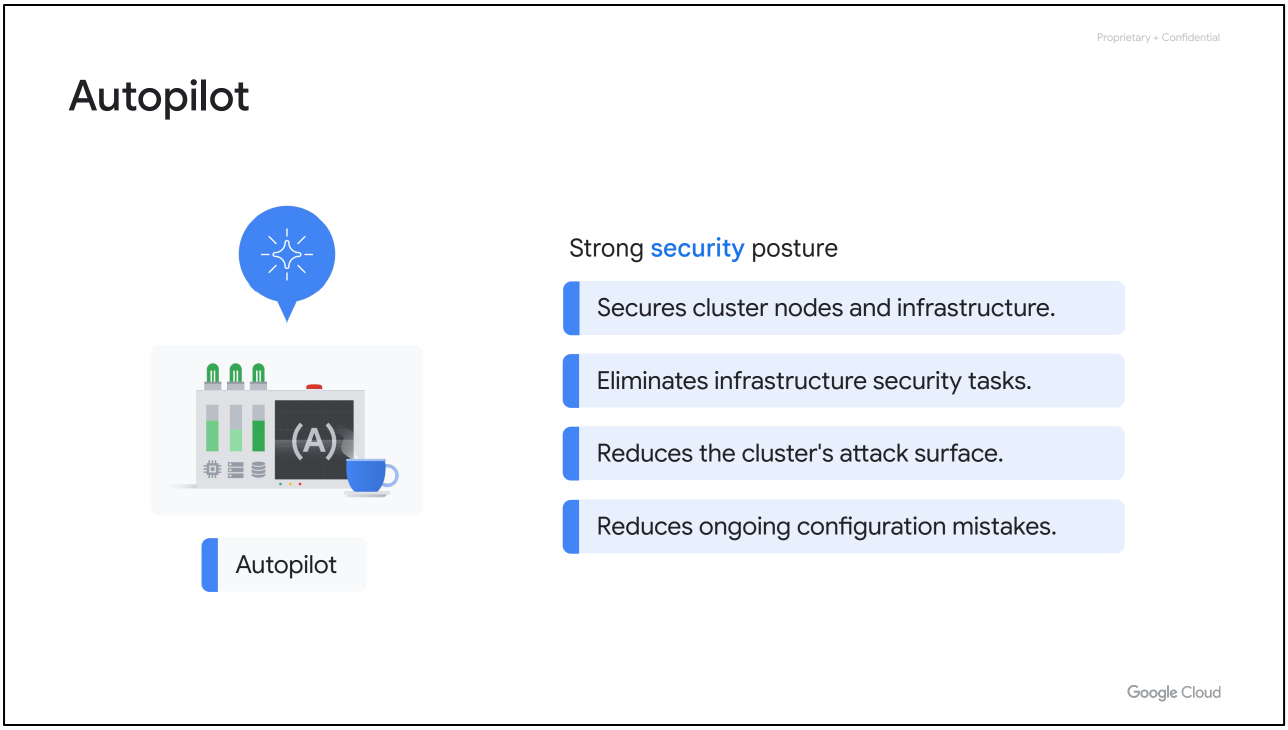
(A)
Autopilot

Optimized for production

- Create clusters according to best practices.
- Defines machine type based on workloads.
- Deploy production-ready GKE clusters faster.

SAY: Let’s examine the benefits and functionality of Autopilot in more detail.

- Autopilot is optimized for **production**. As a Google-managed and optimized GKE instance, the job of Autopilot is to create clusters according to battle-tested and hardened best practices.
- Autopilot defines the underlying machine type for your cluster based on workloads, which optimizes both usage and cost for the cluster and adapts to changing workloads.
- And without the cluster management overhead, Autopilot lets you deploy production-ready GKE clusters faster.



Autopilot

Autopilot

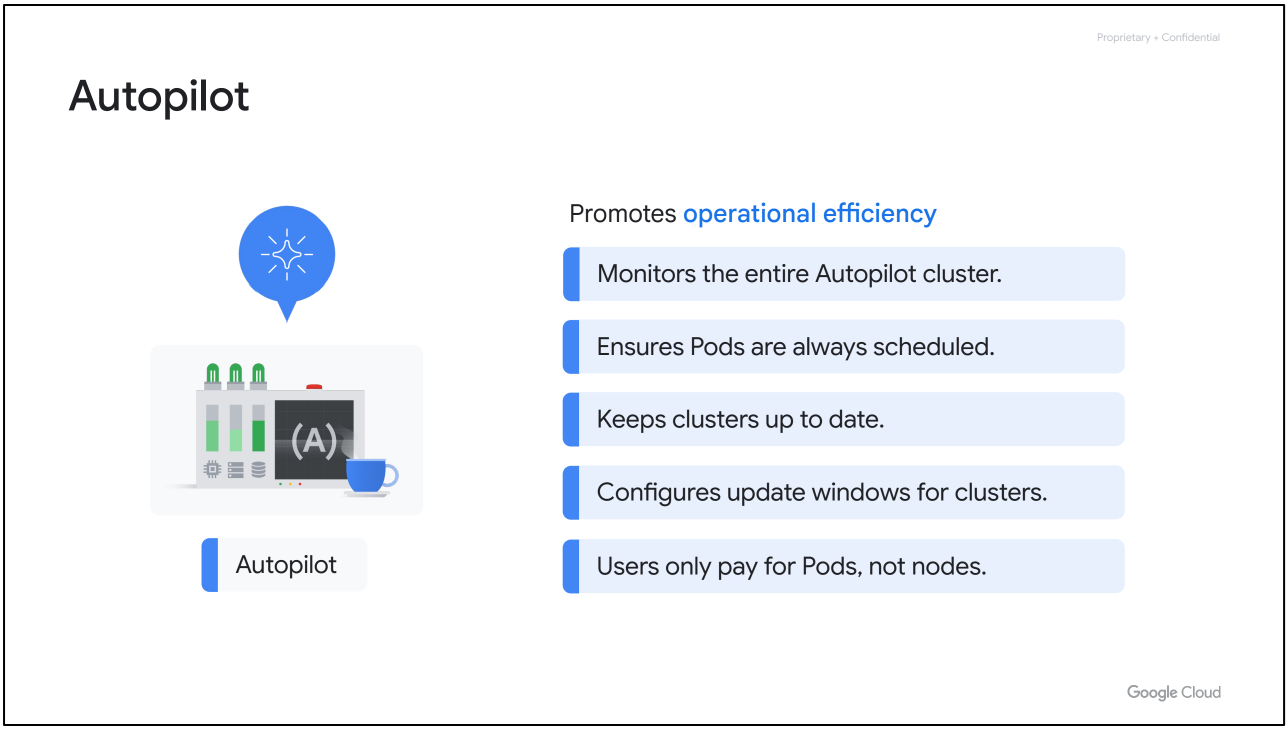
Strong **security** posture

- Secures cluster nodes and infrastructure.
- Eliminates infrastructure security tasks.
- Reduces the cluster's attack surface.
- Reduces ongoing configuration mistakes.

Google Cloud

SAY: Autopilot also helps produce a strong security posture.

Google helps secure the cluster nodes and infrastructure, and it eliminates infrastructure security management tasks. By locking down nodes, Autopilot reduces the cluster's attack surface and ongoing configuration mistakes.



Proprietary + Confidential

Autopilot

Autopilot

Promotes **operational efficiency**

- Monitors the entire Autopilot cluster.
- Ensures Pods are always scheduled.
- Keeps clusters up to date.
- Configures update windows for clusters.
- Users only pay for Pods, not nodes.

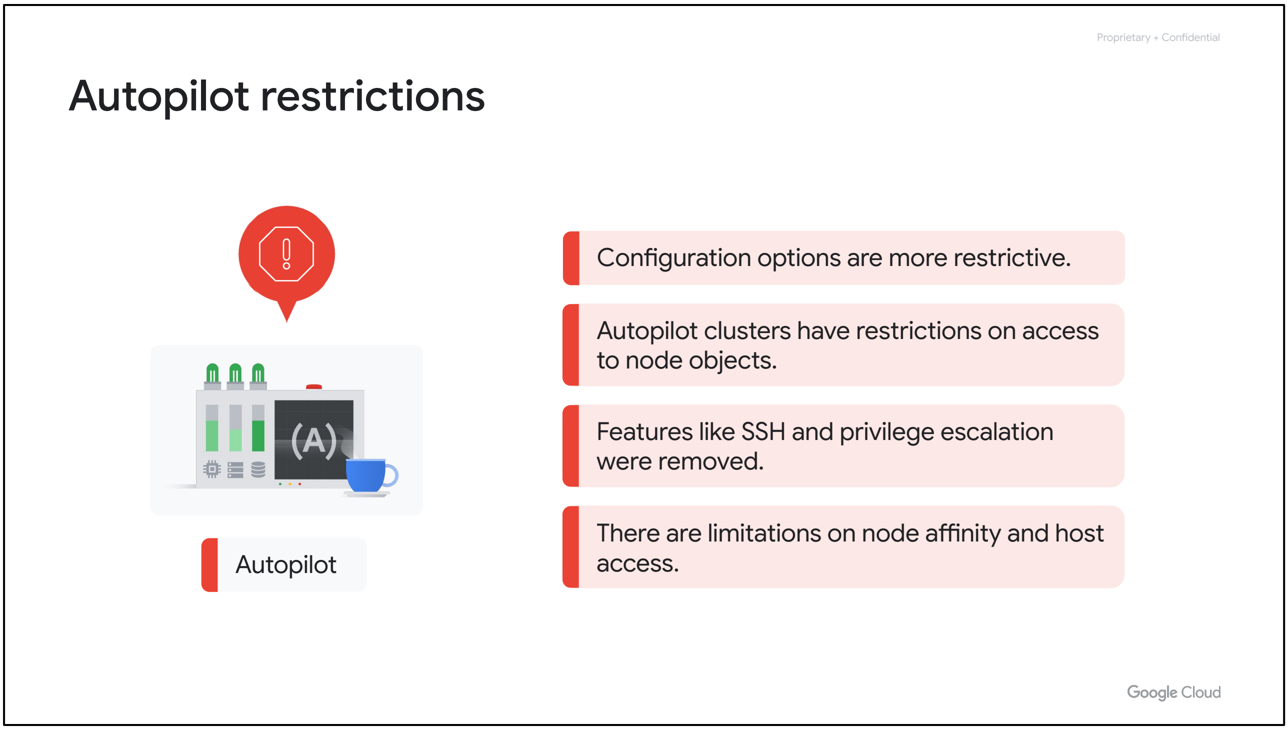
Google Cloud

SAY: Autopilot promotes operational efficiency.

Google monitors the entire Autopilot cluster, including control plane, worker nodes and core Kubernetes system components. This way, it ensures that Pods are always scheduled.

This level of monitoring allows Google to always keep clusters up to date. Autopilot also provides a way to configure update windows for clusters to ensure minimal disruption to workloads.

With Autopilot, Google is fully responsible for optimizing resource consumption. This means you only pay for Pods, not nodes.



Autopilot restrictions

Autopilot

- Configuration options are more restrictive.
- Autopilot clusters have restrictions on access to node objects.
- Features like SSH and privilege escalation were removed.
- There are limitations on node affinity and host access.

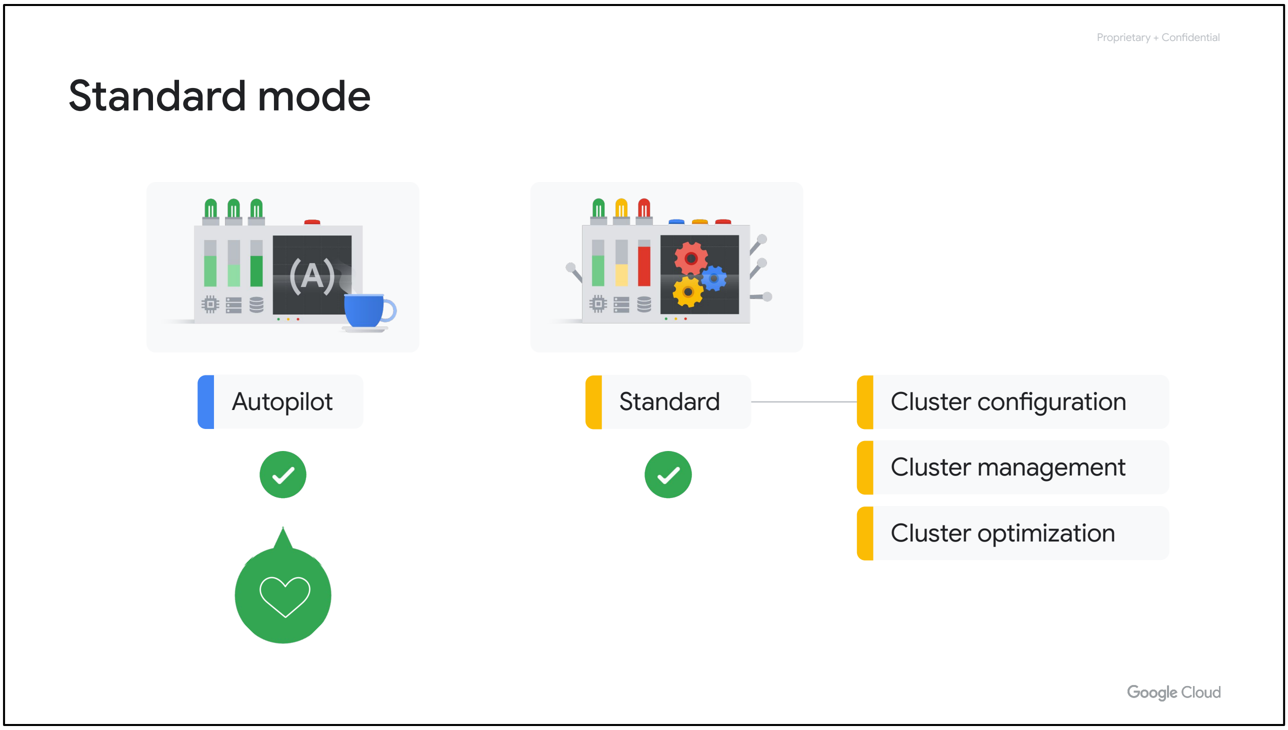
Google Cloud

SAY: Now that you've seen many of the Autopilot mode benefits, let's look at some **restrictions** presented with this operation mode.

The configuration options in GKE Autopilot are more restrictive than in GKE Standard. This is because GKE Autopilot is a fully managed service and has a Pod-scheduling service level agreement.

Autopilot clusters also have restrictions on access to node objects. Features like SSH and privilege escalation were removed and there are limitations on node affinity and host access.

However, all Pods in GKE Autopilot are scheduled with a Guaranteed class Quality of Service (or QoS). But this requires a minor configuration change.



Standard mode

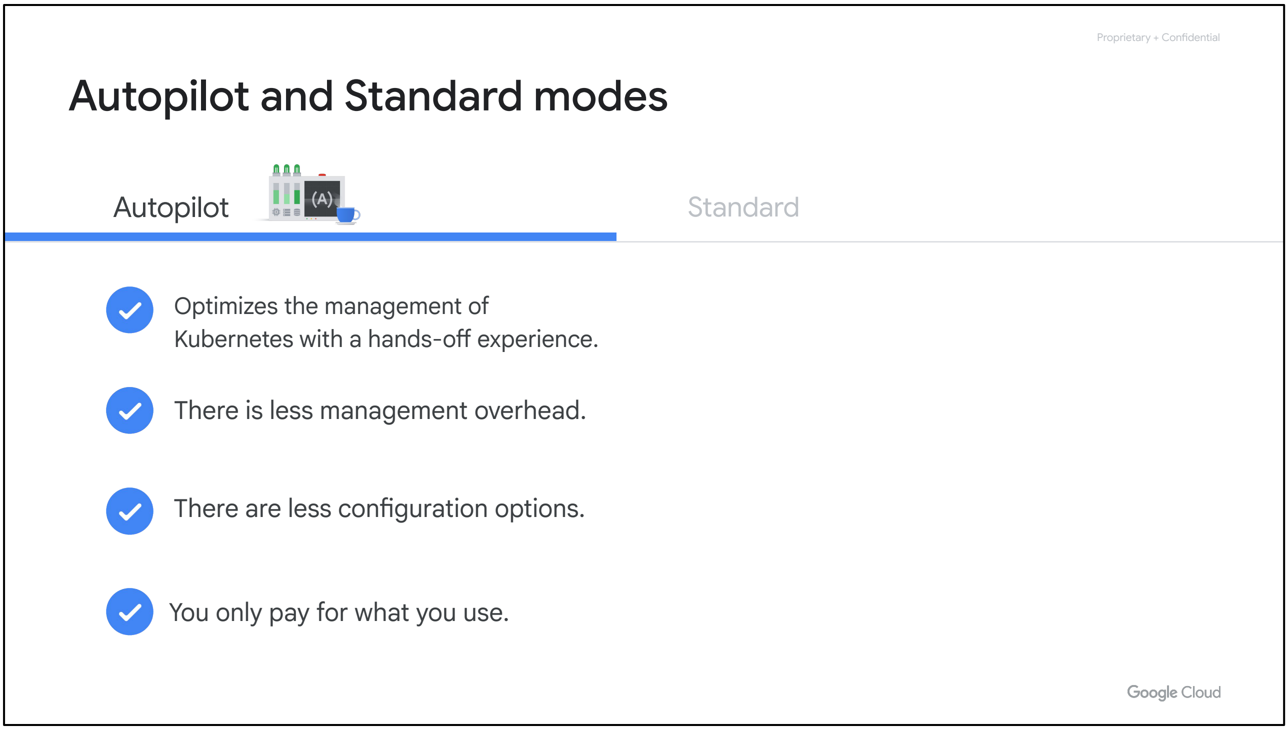
Autopilot ✓

Standard ✓

- Cluster configuration
- Cluster management
- Cluster optimization

SAY: The GKE Standard mode has the same functionality as Autopilot, but **you're** responsible for the configuration, management, and optimization of the cluster.

Unless you require the specific level of configuration control offered by GKE standard, it's recommended that you use Autopilot mode.



Proprietary + Confidential

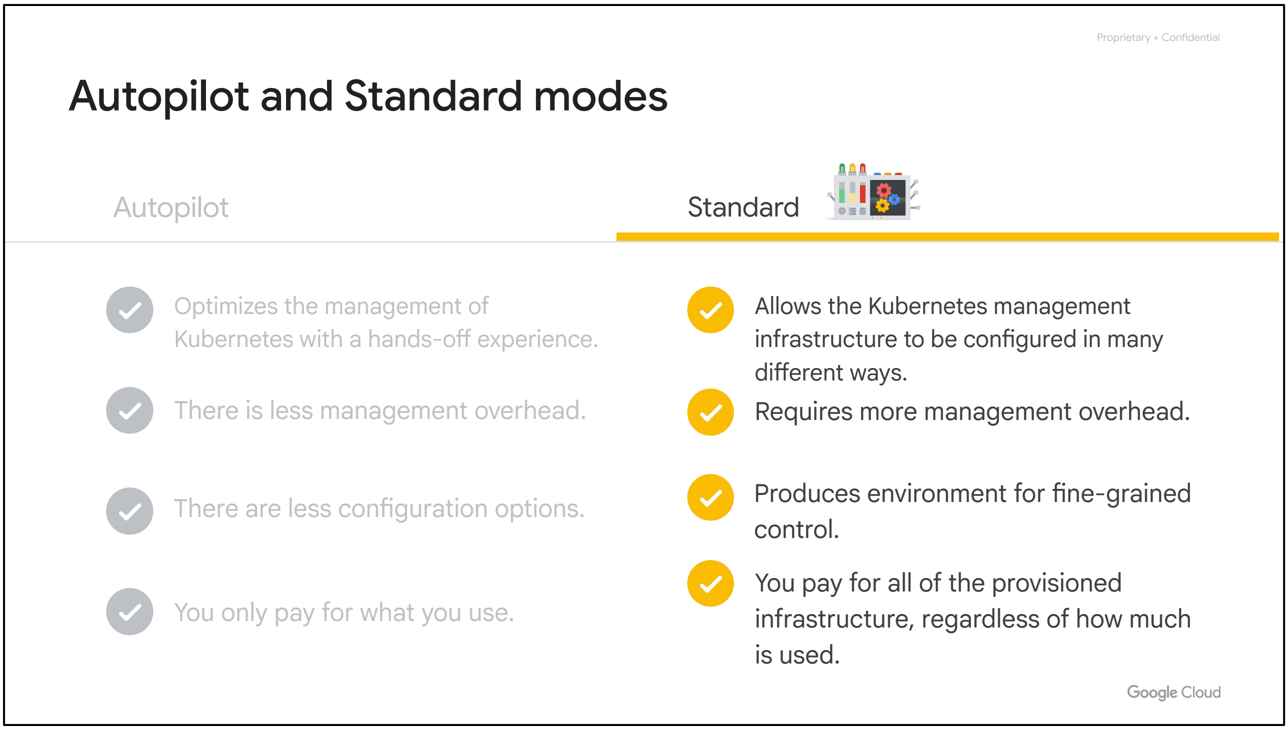
Autopilot and Standard modes

Autopilot | Standard

- ✓ Optimizes the management of Kubernetes with a hands-off experience.
- ✓ There is less management overhead.
- ✓ There are less configuration options.
- ✓ You only pay for what you use.

Google Cloud

SAY: At a high level, **Autopilot** mode optimizes the management of Kubernetes with a hands-off experience. However, less management overhead means less configuration options. And with Autopilot GKE, you only pay for what you use.



Autopilot and Standard modes

Autopilot

- ✓ Optimizes the management of Kubernetes with a hands-off experience.
- ✓ There is less management overhead.
- ✓ There are less configuration options.
- ✓ You only pay for what you use.

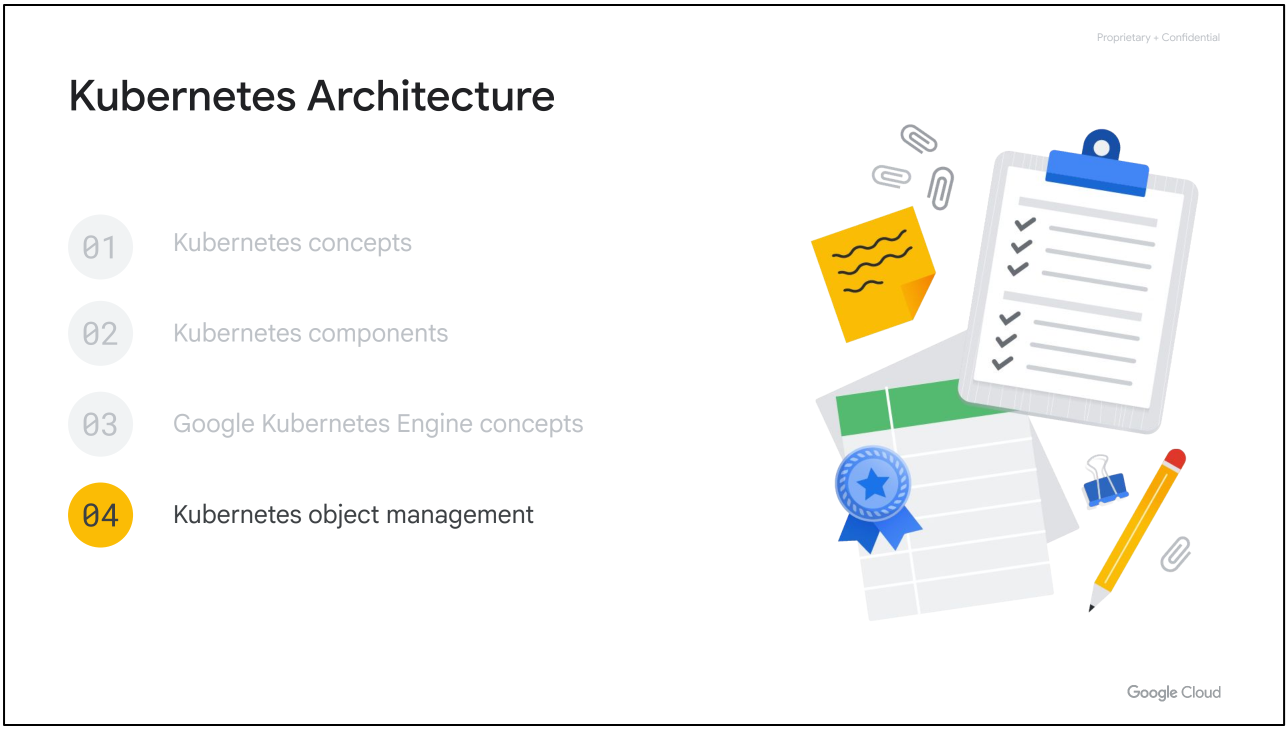
Standard

- ✓ Allows the Kubernetes management infrastructure to be configured in many different ways.
- ✓ Requires more management overhead.
- ✓ Produces environment for fine-grained control.
- ✓ You pay for all of the provisioned infrastructure, regardless of how much is used.

Google Cloud

SAY: The **Standard mode** allows the Kubernetes management infrastructure to be configured in many different ways. This requires more management overhead, but produces an environment for fine-grained control.

With GKE Standard, you pay for all of the provisioned infrastructure, regardless of how much is used.



Kubernetes Architecture

01 Kubernetes concepts

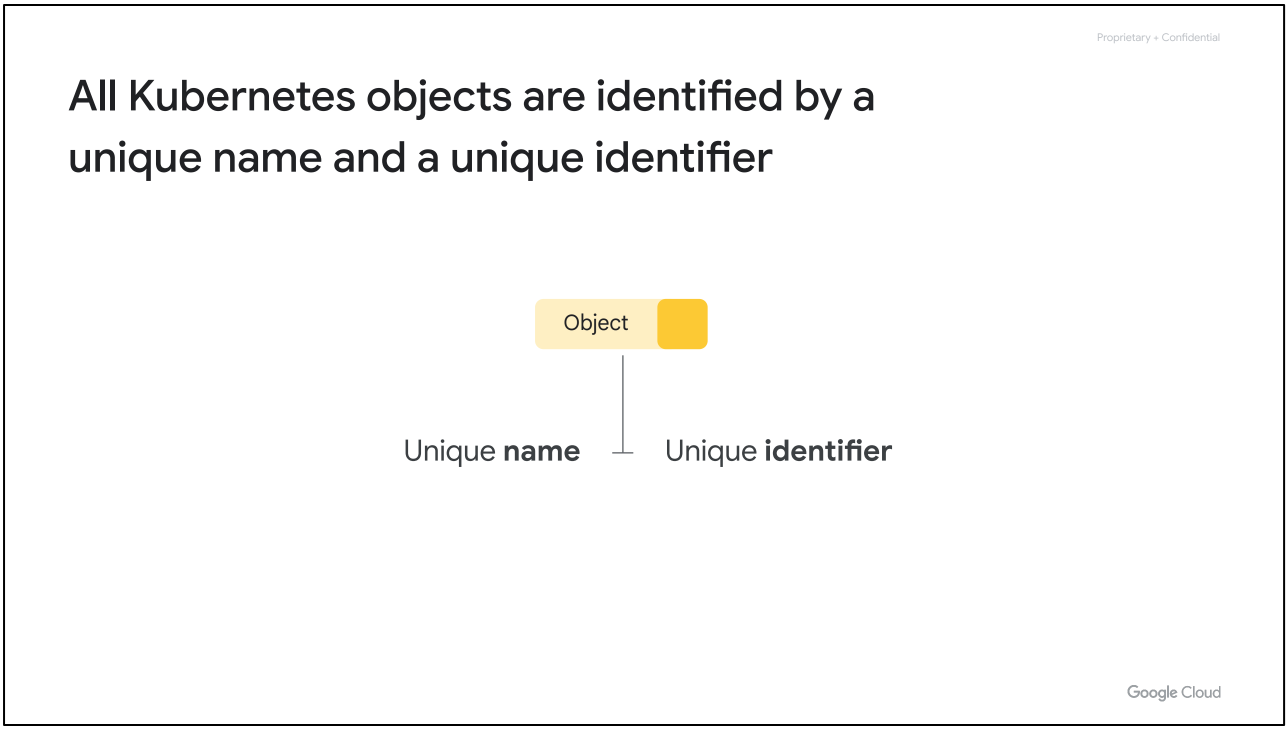
02 Kubernetes components

03 Google Kubernetes Engine concepts

04 Kubernetes object management

Google Cloud

SAY: Let’s finish this section of the course by exploring **Kubernetes object management**.



All Kubernetes objects are identified by a unique name and a unique identifier

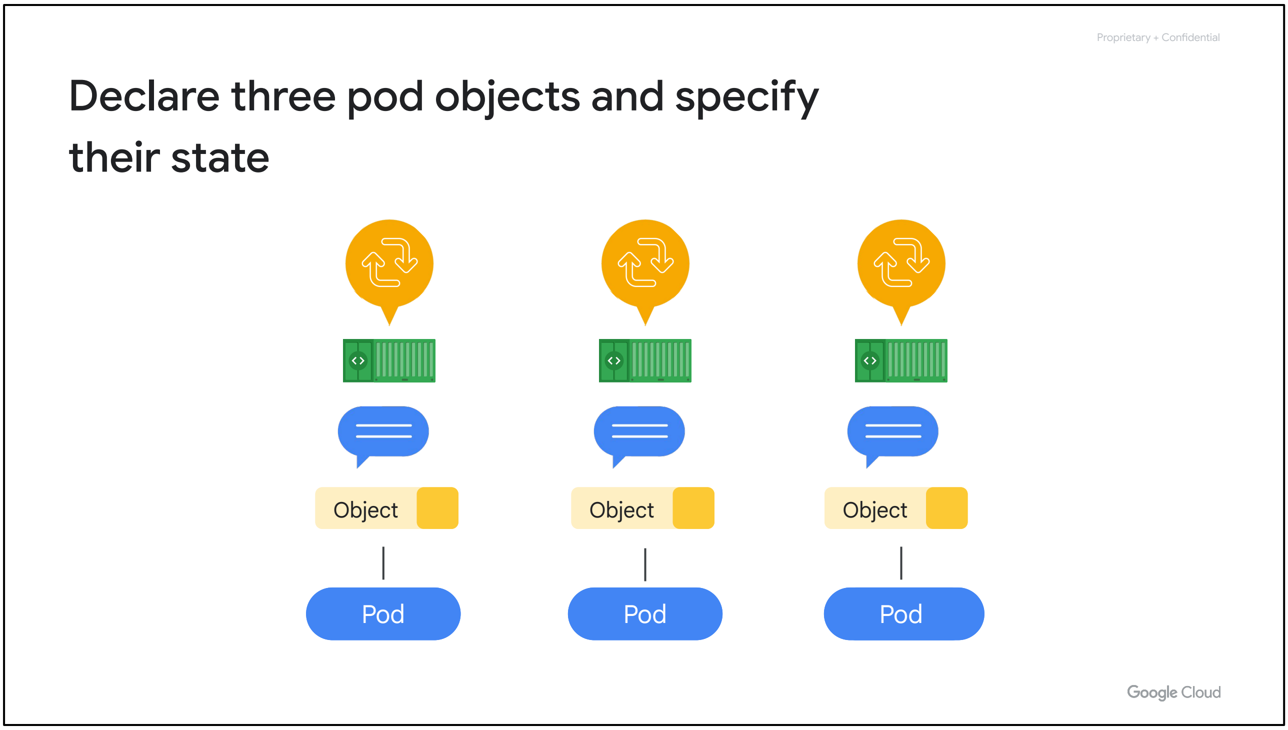
Object

Unique **name**

Unique **identifier**

Google Cloud

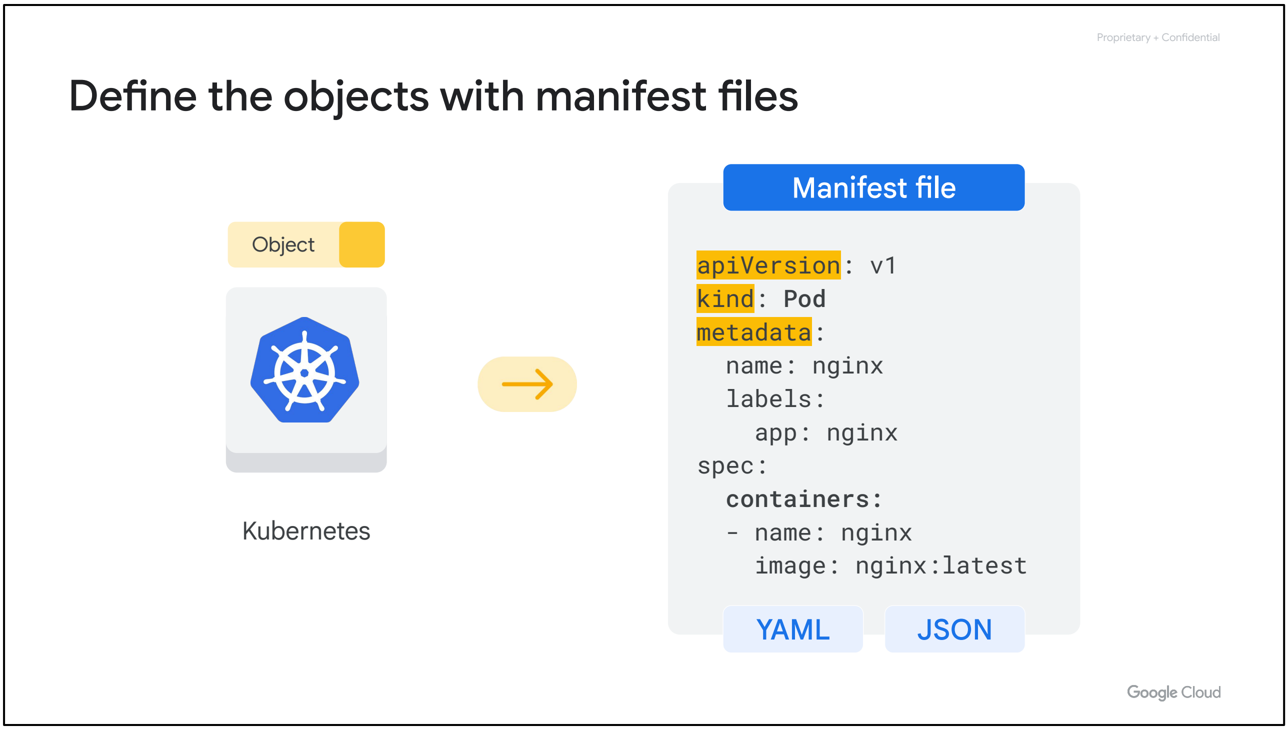
SAY: It’s important to know that all Kubernetes objects are identified by a unique *name* and a unique *identifier*.



Declare three pod objects and specify their state

SAY: Recall the earlier example where you wanted the three nginx Web servers to run all the time. The simplest way to achieve this is by declaring three pod objects and specifying their state. For each, a Pod must be created and an nginx container image must be used.

Let’s see how to make this declaration.



Define the objects with manifest files

Object

Kubernetes

Manifest file

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

YAML

JSON

Google Cloud

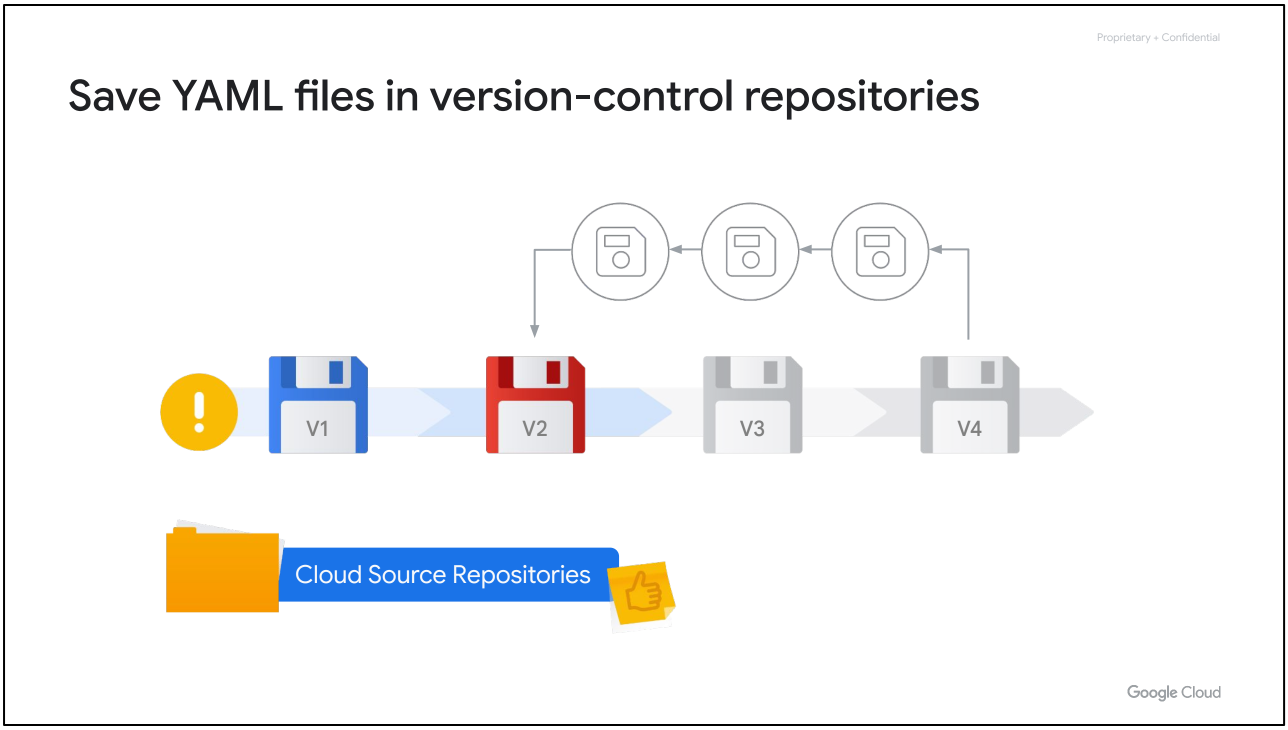
SAY: You define the objects you want Kubernetes to create and maintain with manifest files. These are ordinary text files that can be written in YAML or JSON. YAML is used in this course.

This manifest file defines a desired state for a Pod: its name and a specific container image for it to run.

Required fields include:

- **apiversion**, which states the Kubernetes API version used to create the object
- **kind,** which states the object you want (in this case, a Pod)
- And **metadata**, which identifies the object name, unique ID, and an optional namespace

If several objects are related, it’s a best practice to define them all in the same YAML file. This makes things easier to manage.

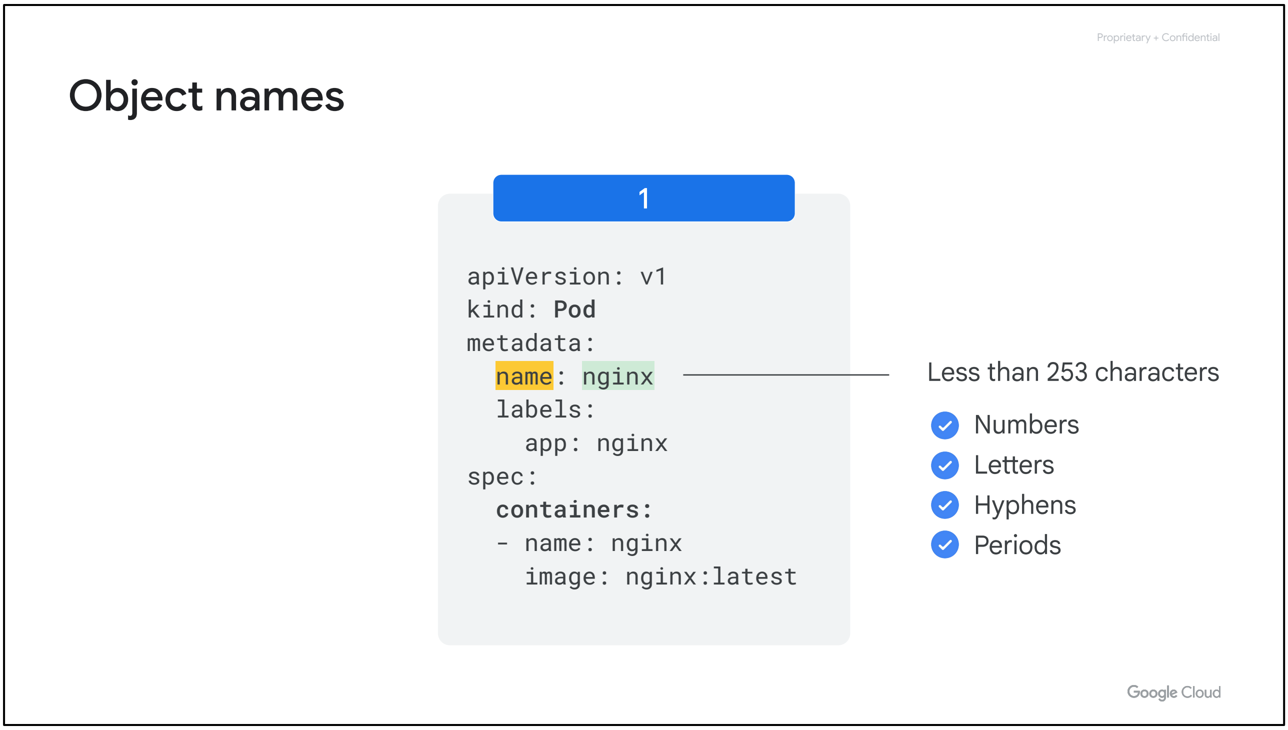


Save YAML files in version-control repositories

Google Cloud

SAY: We strongly recommend saving YAML files in version-control repositories so it’s easier to track and manage changes and to undo those changes when necessary.

It’s also helpful when recreating or restoring a cluster. *Cloud Source Repositories* is a popular service for this purpose.



Proprietary + Confidential

Object names

1

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

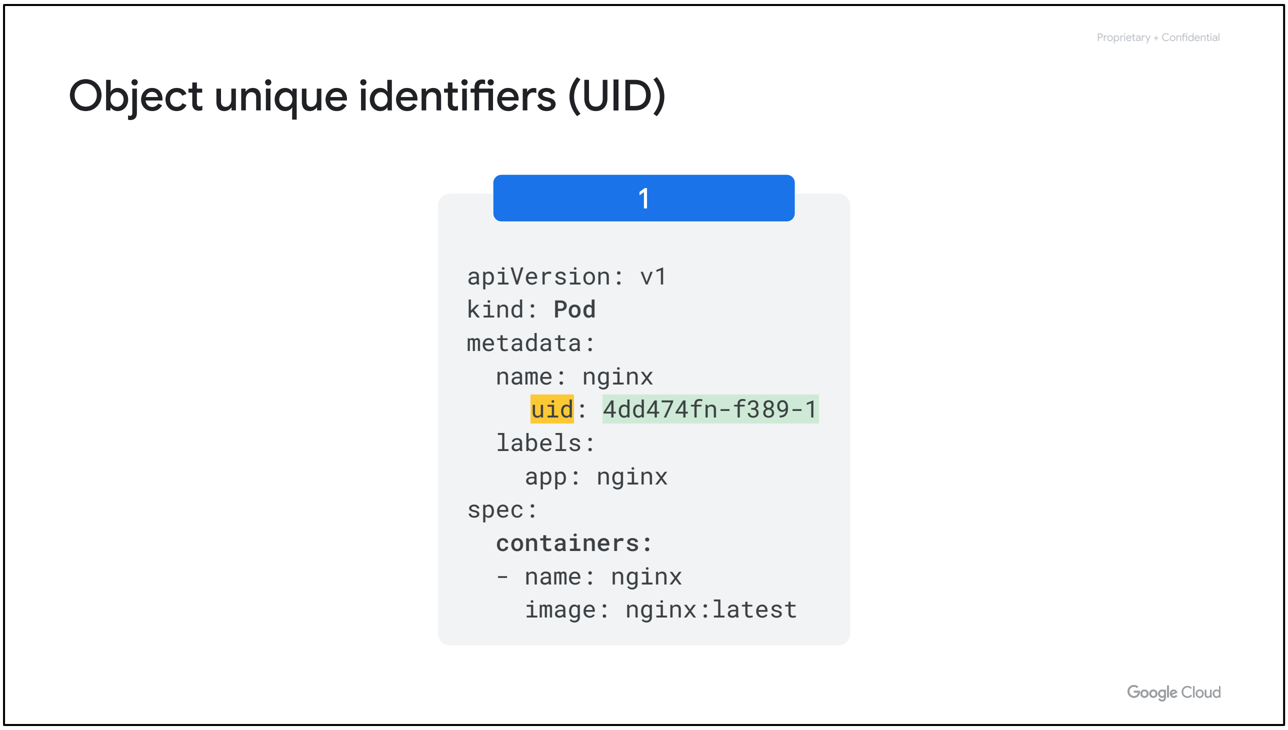
Less than 253 characters

- ✓ Numbers
- ✓ Letters
- ✓ Hyphens
- ✓ Periods

Google Cloud

SAY: Let’s look at the details of an object. First, all objects are identified by a name.

Names must consist of a unique string less than 253 characters. Numbers, letters, hyphens, and periods are allowed. Only one object can have a particular name at the same time in the same Kubernetes namespace. However, after an object is deleted, the name can be reused.

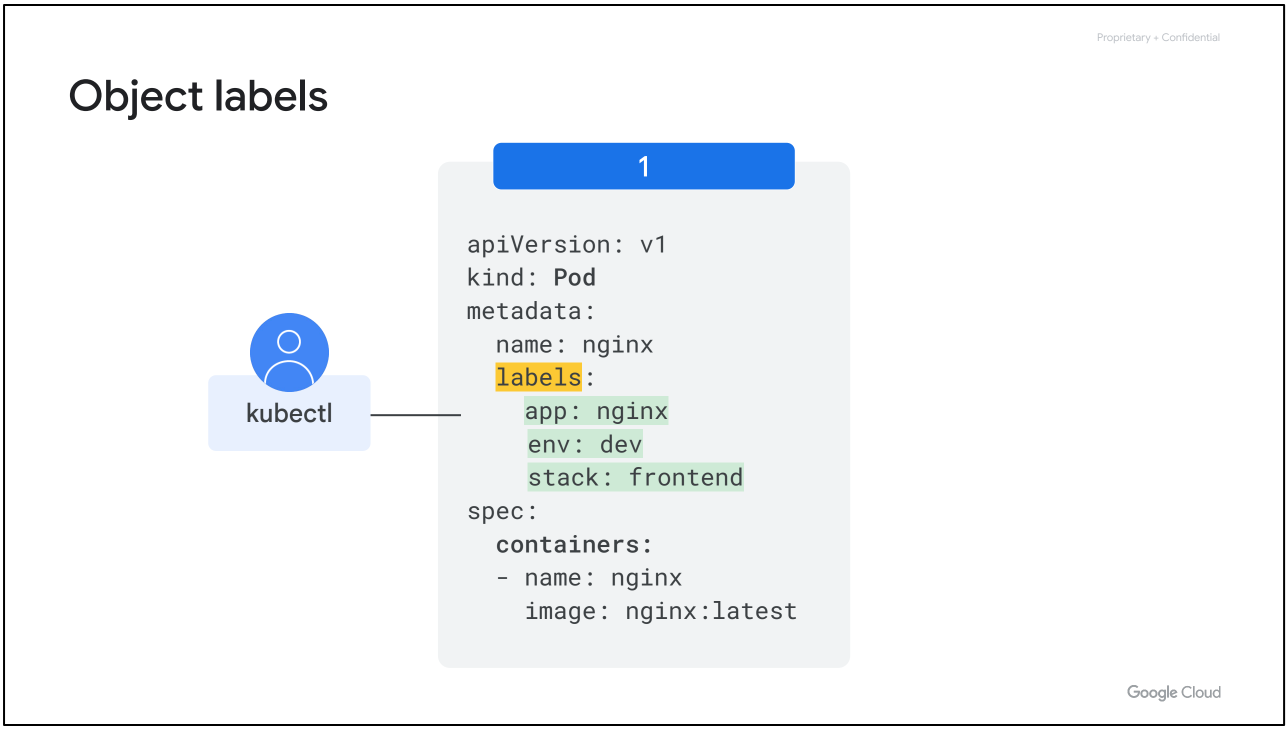


Object unique identifiers (UID)

1

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
    uid: 4dd474fn-f389-1
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

SAY: Second, every object created throughout the life of a cluster has a unique identifier, or UID, generated by Kubernetes.



Proprietary + Confidential

Object labels

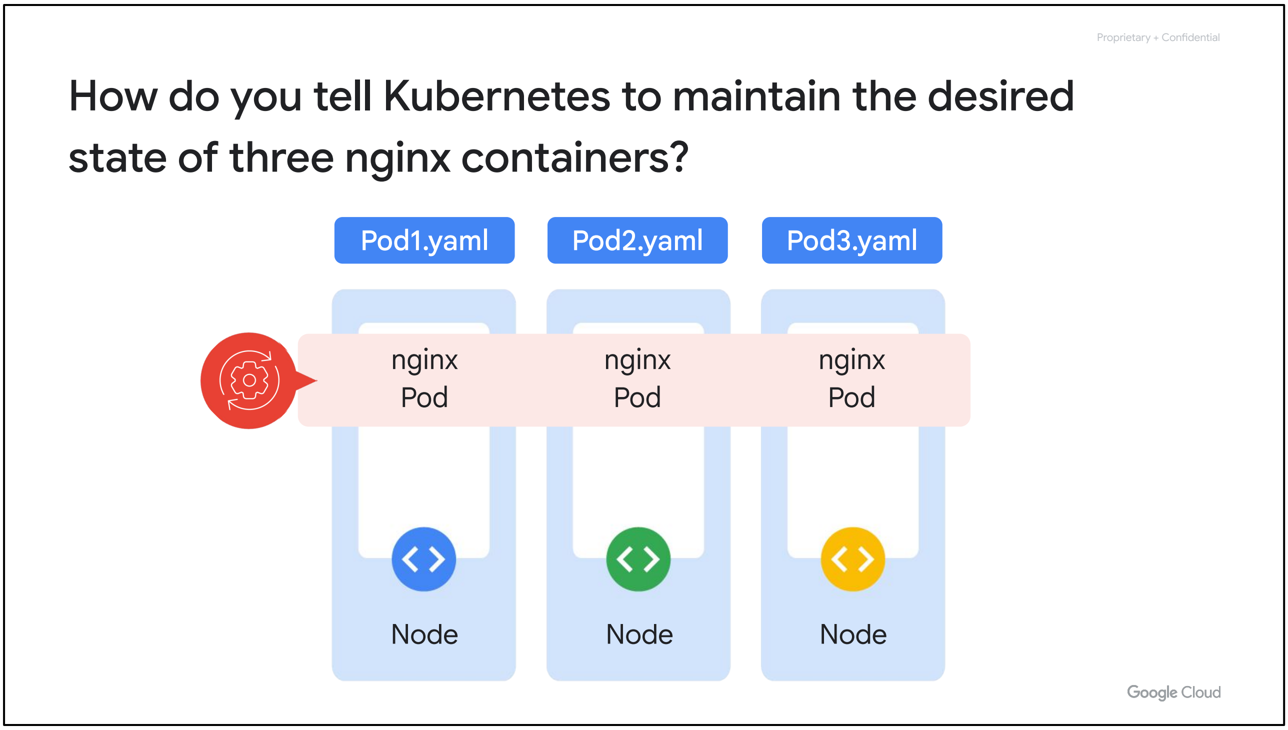
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
    env: dev
    stack: frontend
spec:
  containers:
  - name: nginx
    image: nginx:latest
```

Google Cloud

SAY: And third, there are labels. **Labels** are key-value pairs to tag objects during or after their creation. Labels help identify and organize objects.

A way to select Kubernetes resources by label is through the **kubectl** command. For example, this command can select all the Pods that contain a label called “app” with a value of “nginx.”

Label selectors can be used to ask for all the resources that have a certain value for a label, all those that don’t have a certain value, or even all those with a value in a set you supply.



Proprietary + Confidential

How do you tell Kubernetes to maintain the desired state of three nginx containers?

Pod1.yaml Pod2.yaml Pod3.yaml

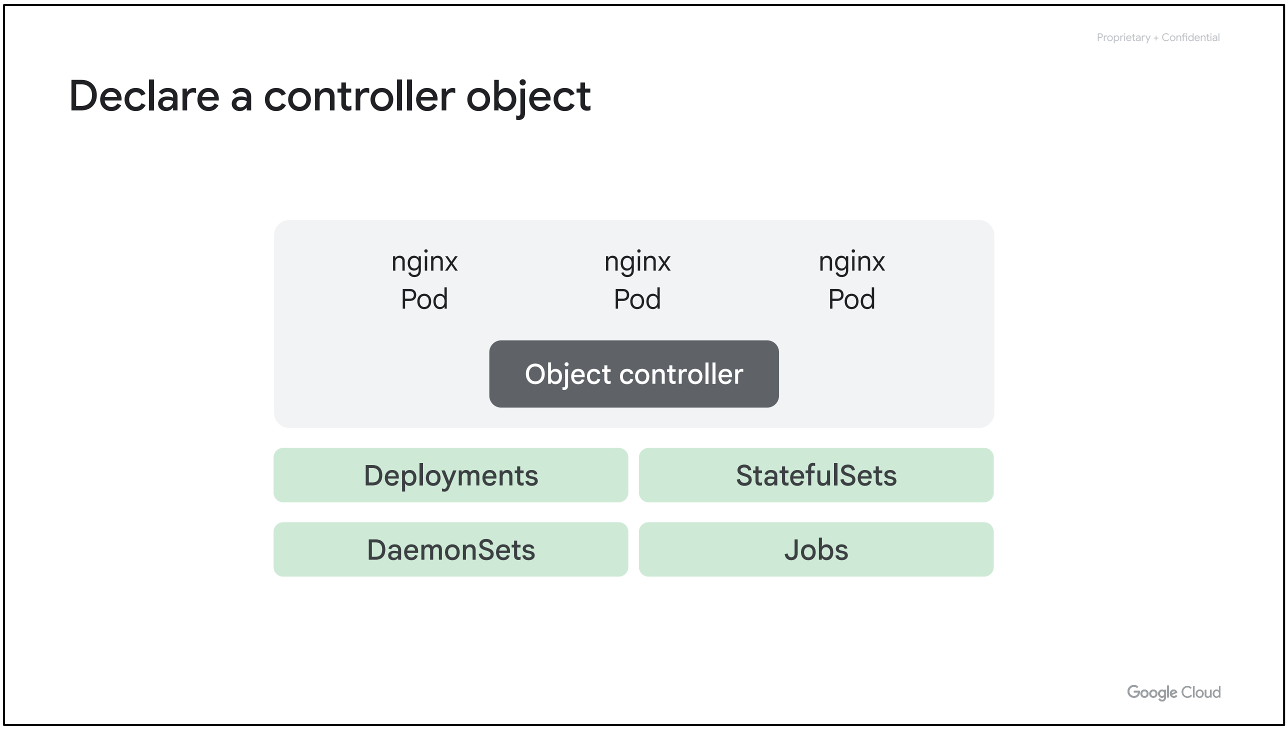
nginx Pod | nginx Pod | nginx Pod

Node | Node | Node

Google Cloud

SAY: Now let’s go back to our example. One way to create three nginx web servers is by declaring three Pod objects, each with its own section of YAML. In Kubernetes, a workload is spread evenly across available nodes by default.

So how do you tell Kubernetes to maintain the desired state of three nginx containers? To maintain an application's high availability, you need a better way to manage it in Kubernetes than specifying individual Pods.



Proprietary + Confidential

Declare a controller object

nginx
Pod

nginx
Pod

nginx
Pod

Object controller

Deployments

StatefulSets

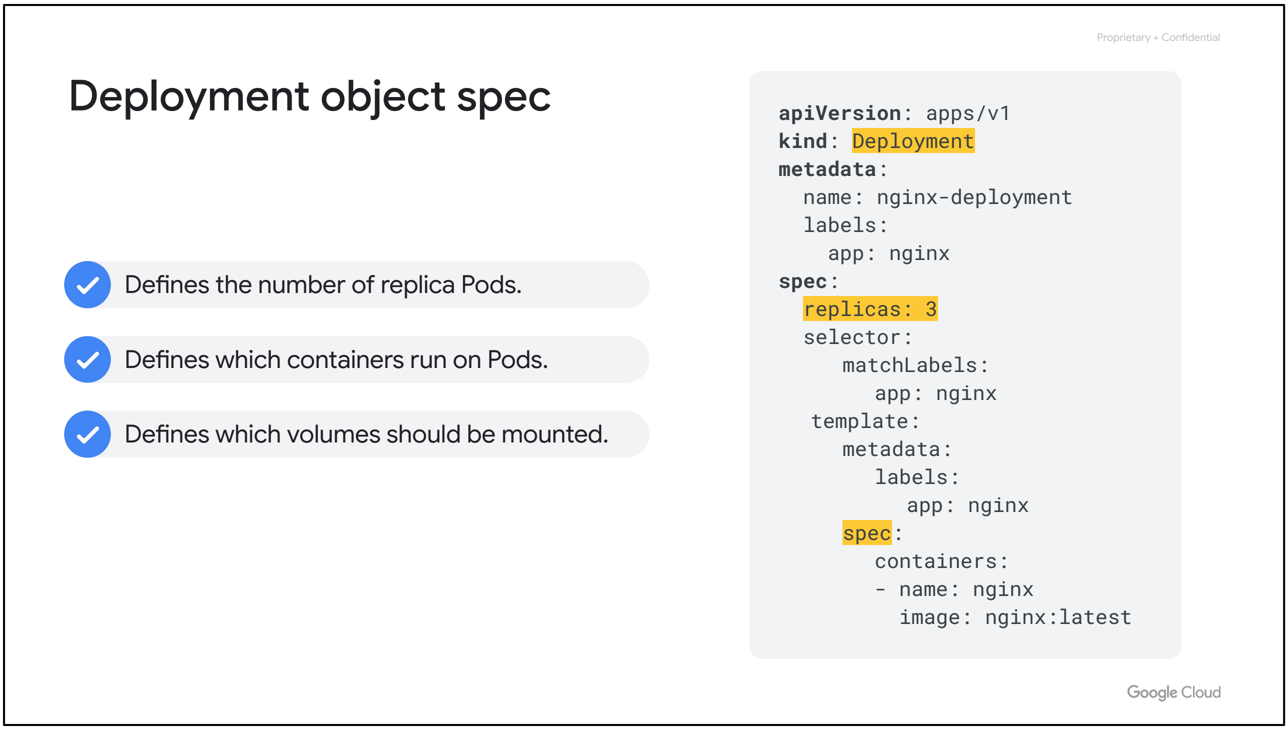
DaemonSets

Jobs

Google Cloud

SAY: One option is to declare a **controller object**. A controller object's job is to manage the state of the Pods. Because Pods are designed to be ephemeral and disposable, they don't heal or repair themselves and are not meant to run forever. Examples include Deployments, StatefulSets, DaemonSets, and Jobs.

Deployments are a great choice for long-lived software components like web servers, especially when you want to manage them as a group.



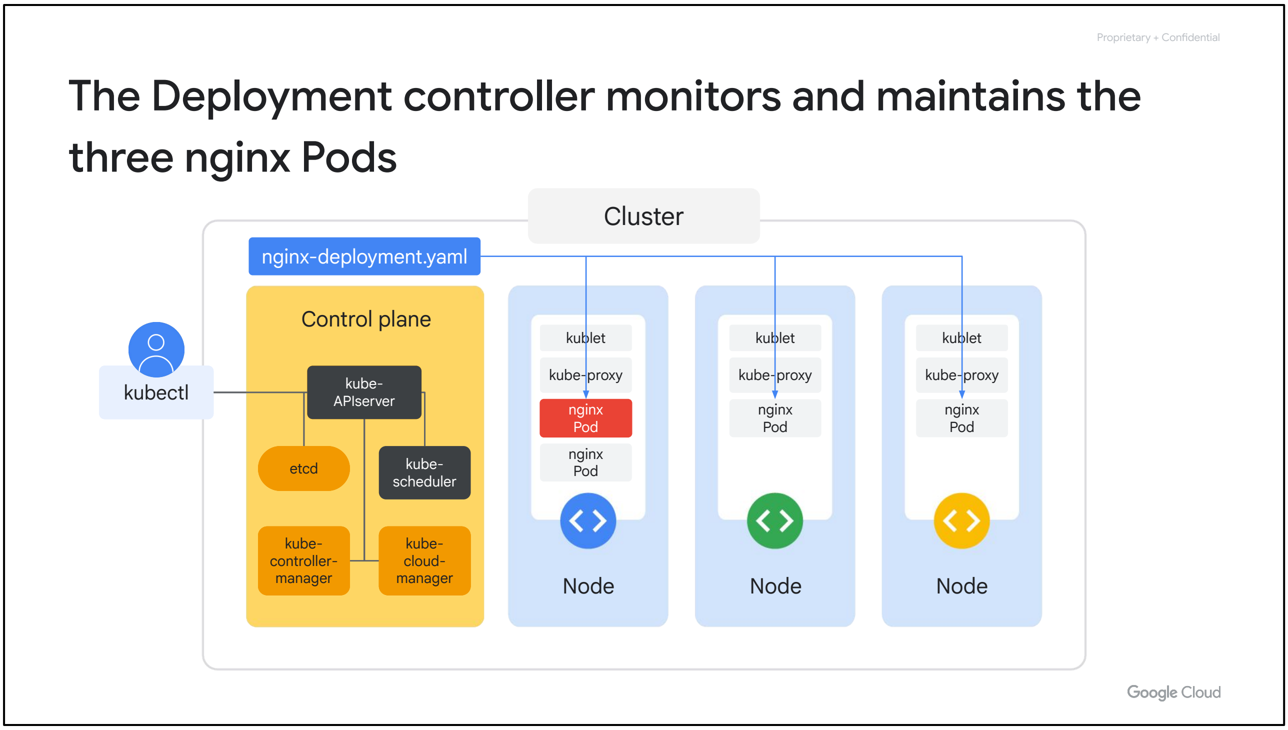
Deployment object spec

- ✓ Defines the number of replica Pods.
- ✓ Defines which containers run on Pods.
- ✓ Defines which volumes should be mounted.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
     matchLabels:
        app: nginx
   template:
     metadata:
        labels:
          app: nginx
     spec:
        containers:
        - name: nginx
          image: nginx:latest
```

SAY: Within a deployment object spec, the number of replica Pods, which containers should run the Pods, and which volumes should be mounted are defined.

Based on these templates, controllers maintain the Pod’s desired state within a cluster.



Proprietary + Confidential

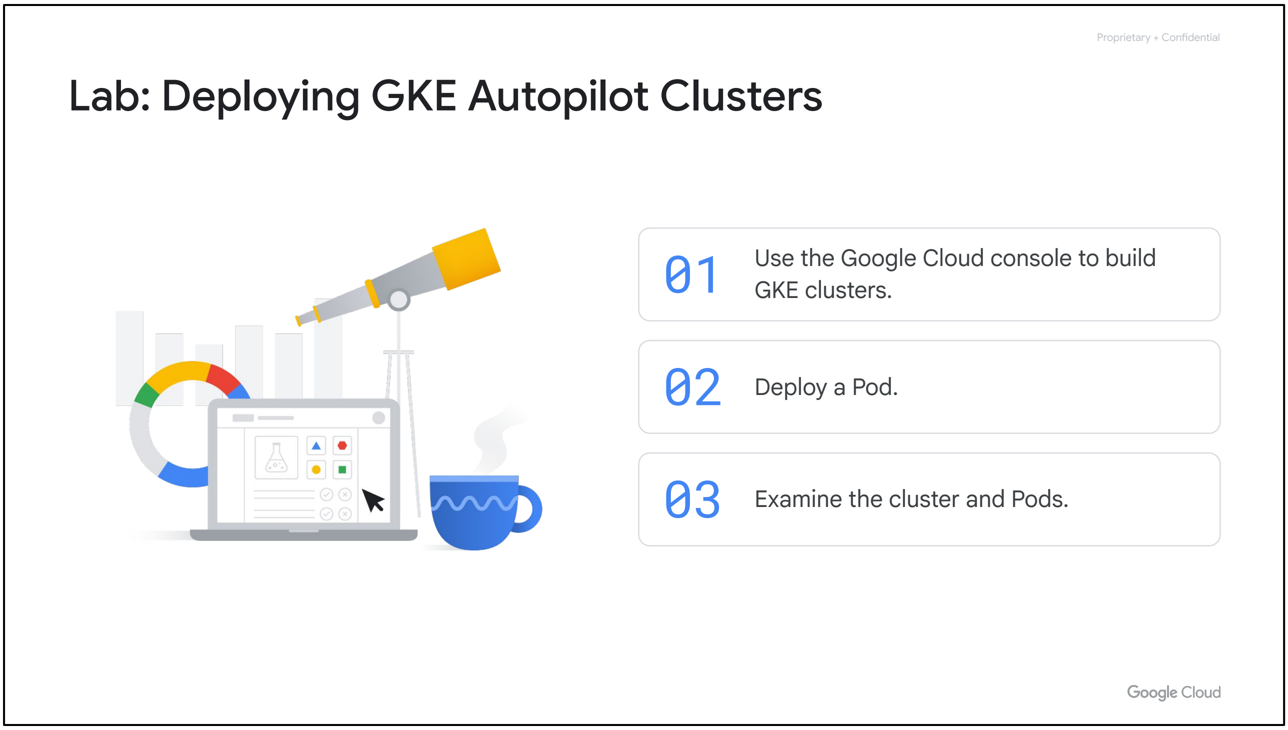
The Deployment controller monitors and maintains the three nginx Pods

Google Cloud

SAY: In our example, the practical effect of the Deployment controller is to monitor and maintain the three nginx Pods. When the kube-scheduler schedules Pods for a Deployment, it notifies the kube-apiserver.

The Deployment controller creates a child object, a **ReplicaSet**, to launch the desired Pods. If one of these Pods fails, the ReplicaSet controller will recognize the difference between the current state and the desired state and try to fix it by launching a new Pod.

So, this means that instead of using multiple YAML manifests or files for each Pod, you used a single deployment YAML to launch three replicas of the same container.



Proprietary + Confidential

Lab: Deploying GKE Autopilot Clusters

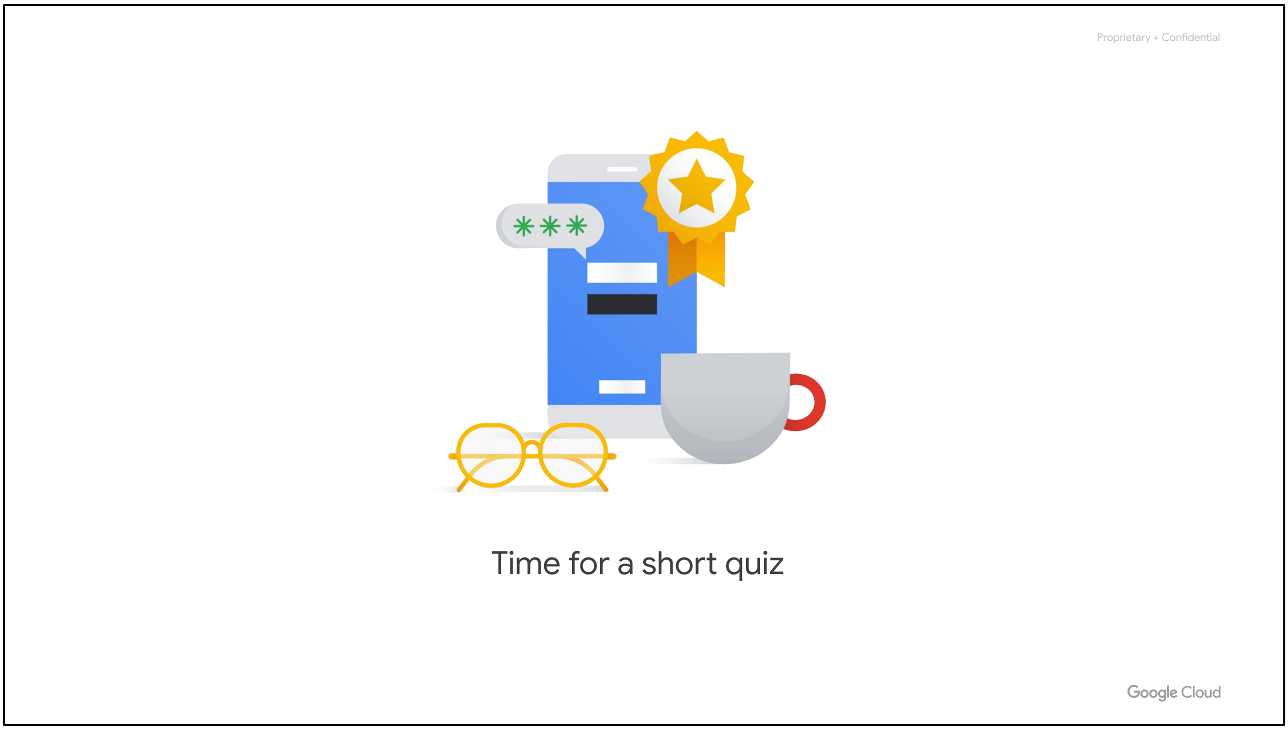
01 Use the Google Cloud console to build GKE clusters.

02 Deploy a Pod.

03 Examine the cluster and Pods.

Google Cloud

SAY: It’s time for some hands-on practice with GKE. In the lab titled “Deploying GKE Autopilot Clusters,” you build and use GKE clusters, then you deploy a sample Pod. Specifically, you use the Google Cloud console to build GKE clusters, deploy a Pod, and then examine the cluster and Pods.

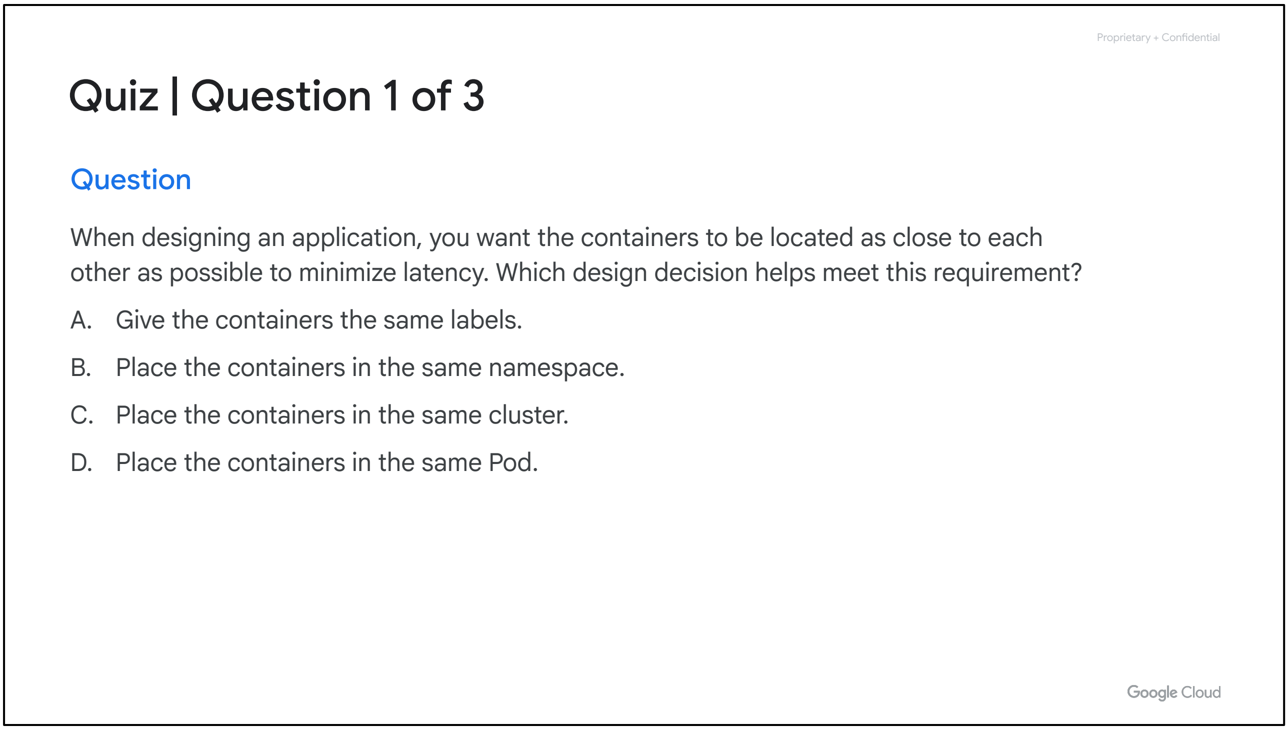


Proprietary + Confidential

Time for a short quiz

Google Cloud

SAY: Let’s pause for a short quiz.



Quiz | Question 1 of 3

Question

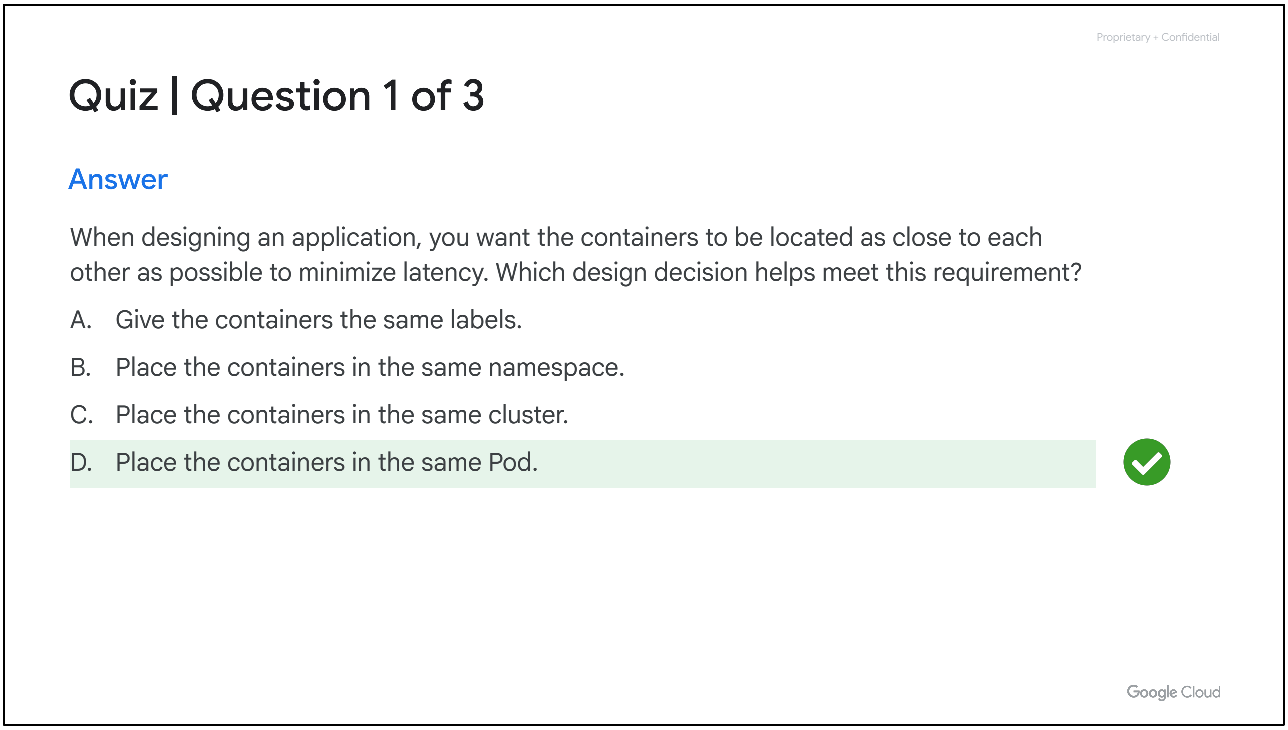
When designing an application, you want the containers to be located as close to each other as possible to minimize latency. Which design decision helps meet this requirement?

A. Give the containers the same labels.

B. Place the containers in the same namespace.

C. Place the containers in the same cluster.

D. Place the containers in the same Pod.



Quiz | Question 1 of 3

Answer

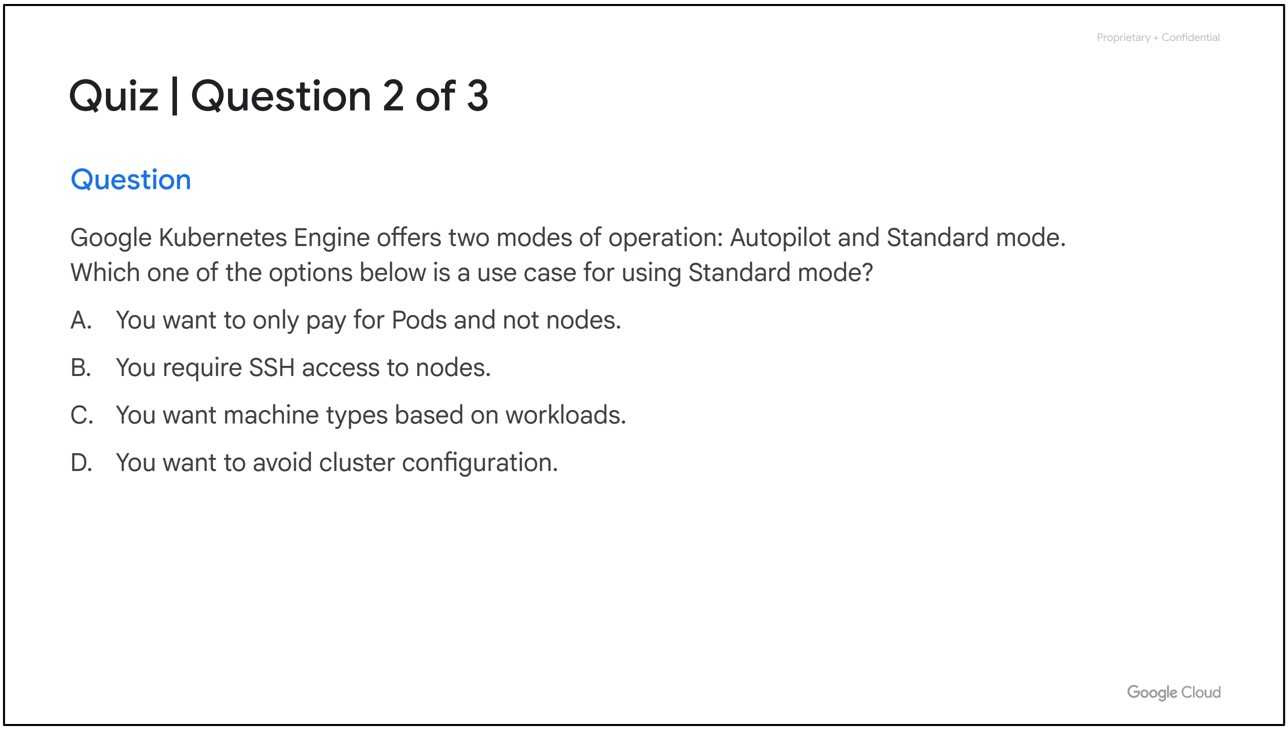
When designing an application, you want the containers to be located as close to each other as possible to minimize latency. Which design decision helps meet this requirement?

A. Give the containers the same labels.

B. Place the containers in the same namespace.

C. Place the containers in the same cluster.

D. Place the containers in the same Pod. ✓



Quiz | Question 2 of 3

Question

Google Kubernetes Engine offers two modes of operation: Autopilot and Standard mode. Which one of the options below is a use case for using Standard mode?

A. You want to only pay for Pods and not nodes.

B. You require SSH access to nodes.

C. You want machine types based on workloads.

D. You want to avoid cluster configuration.

Quiz | Question 2 of 3

Answer

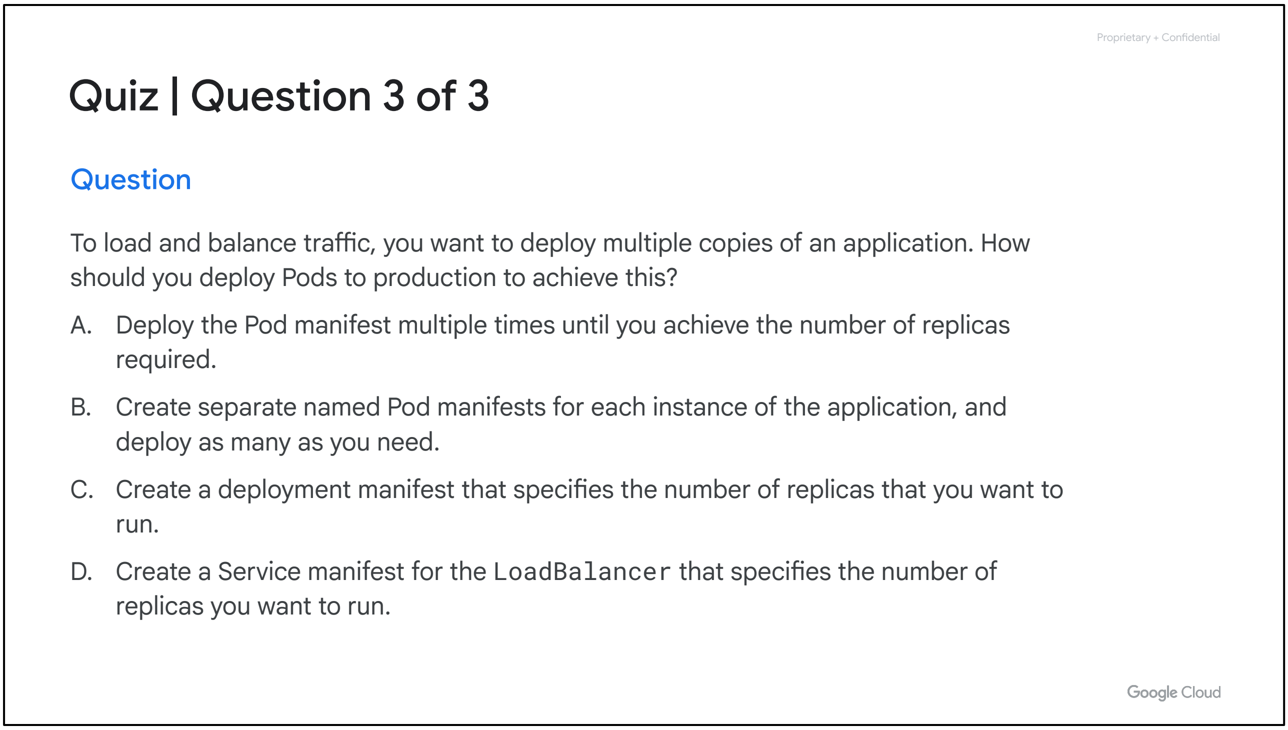
Google Kubernetes Engine offers two modes of operation: Autopilot and Standard mode. Which one of the options below is a use case for using Standard mode?

A. You want to only pay for Pods and not nodes.

B. You require SSH access to nodes. ✓

C. You want machine types based on workloads.

D. You want to avoid cluster configuration.



Quiz | Question 3 of 3

Question

To load and balance traffic, you want to deploy multiple copies of an application. How should you deploy Pods to production to achieve this?

A. Deploy the Pod manifest multiple times until you achieve the number of replicas required.

B. Create separate named Pod manifests for each instance of the application, and deploy as many as you need.

C. Create a deployment manifest that specifies the number of replicas that you want to run.

D. Create a Service manifest for the `LoadBalancer` that specifies the number of replicas you want to run.

Quiz | Question 3 of 3

Answer

To load and balance traffic, you want to deploy multiple copies of an application. How should you deploy Pods to production to achieve this?

A. Deploy the Pod manifest multiple times until you achieve the number of replicas required.

B. Create separate named Pod manifests for each instance of the application, and deploy as many as you need.

C. Create a deployment manifest that specifies the number of replicas that you want to run. ✓

D. Create a Service manifest for the `LoadBalancer` that specifies the number of replicas you want to run.